HARDWARE LANGUAGES AND PROOF

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2011

By
Dominic Anthony Richards
School of Computer Science

Contents

List of Figures

Abstract

HARDWARE LANGUAGES AND PROOF: *A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy by Dominic Richards on 3rd April, 2011.*

Formal methods play a significant and increasing role in hardware verification, but their effectiveness can be impaired by the *ac hoc* nature of mainstream hardware languages such as VHDL, Verilog and SystemC, which have convoluted semantics that often necessitate contrived proof techniques. This dissertation investigates the application of formal reasoning to hardware architectures expressed in an alternative class of *semantically elegant* languages, which support efficient design, whilst also having been developed with proof techniques in mind.

A network-on-chip architecture belonging to the SpiNNaker many-core processor is specified in Concurrent Haskell, and a hand proof is presented which verifies a novel routing mechanism by mathematical induction.

A subset of Bluespec SystemVerilog (BSV) is embedded in the higher order logic of the PVS theorem prover. Owing to the clean semantics of BSV, application of monadic techniques leads to a surprisingly elegant embedding, in which hardware designs are translated into logic *almost verbatim*, preserving types and language constructs. Proof strategies are written in the PVS strategy language; these automatically verify temporal logic theorems concerning the resulting monadic expressions, by employing a combination of model checking and deductive reasoning. The subset of BSV which is embedded includes module definition and instantiation, methods, implicit conditions, scheduling attributes, and rule composition using methods from instantiated modules.

The aforementioned subset of BSV is also embedded in the specification language of the SAL model checker, and a verification strategy is presented which combines the specialised model checking capabilities of SAL with the diverse proof strategies of PVS.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://www.campus.manchester.ac.uk/medialibrary/policies/intellectual-property.pdf`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.manchester.ac.uk/library/aboutus/regulations`) and in The University's policy on presentation of Theses.

Acknowledgments

Thanks to David Lester for many engaging discussions, and for encouraging a spirit of independence in research. Thanks also to Steve Furber, the SpiNNaker team and the wider Advanced Processor Technologies group for a great deal of good advice. Finally, a special mention to Andrew Bardsley, for sharing his extensive knowledge of programming languages and enthusiasm for Haskell.

To my family, for love and encouragement.

Chapter 1

Introduction

> *Without major breakthroughs, design verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry. There is hope for breakthroughs to emerge via a shift from* ad hoc *verification methods to more structured, formal processes.*
>
> 2009 International Technology Roadmap for Semiconductors

> *The significant problems we face today cannot be solved at the same level of thinking we were at when we created them.*
>
> Albert Einstein

1.1 An Explosion of Complexity

In 1965 – when Integrated Circuit (IC) design was in its infancy and chips contained tens of transistors – Gordon Moore made a bold prediction: the number of transistors that could be fabricated on a chip would grow exponentially, doubling every year for at least the next decade [Moo65]. At the time he was criticised for "unrealistic optimism", but his prediction of exponential growth (with an adjusted time period of $1\frac{1}{2}$ – 2 years) held true for long enough to be termed "Moore's law" in the early 1970s and continues to hold today, an astonishing 45 years later. Those 45 years have been the dawn of the information age: commoditised computing; unprecedented connectivity; surging advances across the sciences ... and all of this fuelled by increasingly powerful – and complex – silicon chips. When Moore made his prediction, chips contained tens of transistors: today, industry giants such as Intel and AMD churn out processors

containing several billion, and thousands of smaller vendors produce application-specific ICs (ASICs) with transistor counts ranging into the hundreds of millions.

Fabrication technologies now permit such high transistor counts that design innovation is limited by the ability to exploit such potential complexity by arranging the available transistors into correctly functioning designs. As fabrication technologies evolved over the past half century, tools and techniques for designing integrated circuits evolved simultaneously, in order to deliver increased design efficiency, and hence enable the realisation of evermore complex designs: for several decades, however, the rate of increase in design productivity has been consistently outstripped by the rate of increase in transistor densities[1], causing escalating design team head-counts and spiralling design costs, which are coming to dominate the economics of IC production. Design non-recurring engineering (NRE) costs now frequently exceed manufacturing NRE costs by an order of magnitude [ITR09] and will likely increase further, as long as Moore's law continues to hold and consumers maintain their appetite for evermore complex devices. In particular, design verification techniques are coming under increasing strain; many ASIC teams now employ two or more verification engineers for each designer [ITR09] and there is a real possibility of a "verification bottleneck" in the near future, where the trend of burgeoning device complexity is halted, not by the limits of fabrication technologies, but by the cost of design verification.

IC manufacturers now find themselves in a competitive environment which is increasingly defined by the metric of design efficiency and, in response, are investing heavily in research and development (R&D) for new design and verification technologies. One champion of this R&D effort has been *formal verification* – a mathematical paradigm which is fundamentally more rigorous than traditional test-based approaches. Formal verification has developed over several decades as a sideline in the hardware industry: however, spurred by the ongoing complexity crisis, it now looks set to join the centre-stage. The 2009 International Technology Roadmap for Semiconductors (ITRS '09) [ITR09] – a definitive industry-wide consensus on R&D priorities – found that 9.4% of design errors in the companies it surveyed were identified with formal or semi-formal[2] techniques, and stipulated that this should increase to 45% over the following 15 years. For this to happen, however, a great

[1] Design productivity (measured as the number of logic gates per design-year) increased at an average rate of 40% per year between 1990 and 2009, but over the same period the number of transistors that could be fabricated onto a chip increased at 40% – 60% per year, in accordance with Moore's Law (historically doubling every $1\frac{1}{2}$ – 2 years) [ITR09].

[2] Combining formal techniques with traditional *ad hoc* approaches [BAWR07].

deal of innovation will be required in the way that formal methods are applied in the hardware design flow. This dissertation investigates one such innovation: the application of formal reasoning to hardware designs expressed in *semantically elegant* languages...

Thesis Contributions

Hardware is typically designed using *ad hoc* languages such as VHDL [IEE94], Verilog [TM96] and SystemC [Gro02], which have evolved organically over the lifetime of the industry and are therefore well integrated with current design flows, but unfortunately sit poorly with proof-based verification techniques owing to their convoluted, and often ambiguous, semantics [BGG+92, Gor95, Klo95, MRH+01]. In contrast, an alternative class of semantically elegant languages allows efficient design, whilst also having been developed with formal verification in mind. This dissertation demonstrates the application of proof techniques to hardware designs expressed in two such languages:

- **Concurrent Haskell –** a language for software design which combines functional programming with channel-based concurrency [PJGF96]. It will be used to develop a behavioural specification of a network-on-chip (NoC), for subsequent verification with hand-proof.

- **Bluespec SystemVerilog (BSV) –** a language for hardware design and synthesis which combines a guarded-action model of concurrency with language constructs from the functional paradigm [Nik04]. A subset of BSV will be embedded in the specification logics of the PVS theorem prover and the SAL model checker, providing access to a wide variety of automated proof techniques.

For both languages, we will find that the combination of functional programming with a clean model of concurrency yields hardware descriptions which are highly amenable to the application of proof. This result is particularly encouraging for BSV, which is a relatively new language that has already been shown to reduce design time when compared to hand-written VHDL or Verilog, whilst producing comparable hardware for many applications [GW08, Nik04, WNRD04] – this dissertation will provide evidence that it is also more amenable to automated reasoning. Furthermore, this insight is timely, because BSV presently has *no* tool support for automated reasoning, placing it very much behind the curve when compared to VHDL and Verilog.

The technical contributions of this thesis are:

- A novel application of lightweight formal methods to the SpiNNaker network-on-chip architecture, which involves behavioural specification in Concurrent Haskell, together with verification by hand-proof.
- A shallow embedding of a subset of Bluespec SystemVerilog in the higher order logic of the PVS theorem prover. A novel application of monadic techniques [Mog89, Wad92b] allows a surprisingly clean embedding, in which BSV designs are translated into logic *almost verbatim*, preserving types and language constructs. The subset which is embedded includes module definition and instantiation, methods, implicit conditions, scheduling attributes, and rule composition using methods from instantiated modules.
- Proof strategies, written in the PVS strategy language, which automatically verify temporal logic theorems concerning the aforementioned monadic expressions, by employing a combination of model checking and deductive reasoning.
- An embedding of the same subset of BSV in the SAL language, which provides access to the wide array of model checking tools in the SAL suite.

Of particular note is the monadic embedding of BSV in the PVS logic. Whilst monads have been used before to address the notoriously messy issue of verifying state-based computation with theorem proving, their application to BSV yields surprisingly clean results. BSV has its roots in a minimalist guarded action language (of the kind used to specify state machines for model checking) and was expanded into a fully-featured hardware language with strong influences from functional programming. As a result, we will see that an environment such as PVS – which combines integrated model checking and theorem proving with a higher order functional specification language – allows the almost verbatim translation of BSV source code to a monadic form which can be directly verified using a combination of model checking and automated deduction.

1.2 Modern Hardware Design

When Gordon Moore made his prediction in the 1960s, hardware designers sat with 'coloured rectangles' and drew transistors by hand on graphical representations of circuits. Through the 1970s and 1980s, transistor counts increased and designers

progressed to *gate level* languages for a large amount of design. These languages abstracted away from individual transistors to the level of logic gates. They hid the detail of transistor-level structure, which made designs more concise and understandable; this increased design productivity by reducing design time, reducing the incidence of errors and making errors easier to spot when they did occur. The new languages were supported by *place and route* tools which automatically compiled gate-level designs down to the transistor level, with a small performance penalty compared to hand-crafted transistor level circuits.

By the mid 1980s, fabrication technologies allowed hundreds of thousands of transistors per chip, and designers added another level of abstraction. Register Transfer Level (RTL) languages provided types such as bounded integers, as well as programming language constructs such as if-then-else statements, case expressions and arithmetic operators. Designs expressed in these languages could be automatically compiled down to gate level using synthesis software. RTL languages gained widespread popularity because they employed simple abstractions to hide the detail of standard circuit components, whilst allowing automatic synthesis of reasonably efficient gate level circuits. Two RTL languages gained widespread acceptance; VHDL [IEE94] and Verilog [TM96].

Alongside the development of RTL languages, tools emerged that raised the level of abstraction further still by synthesizing from high-level imperative languages such as C, C++ and SystemC [Gro02], which allow the functionality of a design to be specified without fully defining the architecture that will produce this functionality. These *high-level synthesis* approaches have seen a strong uptake for some applications, but have not yet seen widespread adoption as a replacement for RTL languages; at present, they do not produce hardware that is consistently competitive with hand-coded RTL (in terms of speed, area and power) for most designs [ITR09, WNRD04].

High-level languages are, however, seeing extensive uptake for another use; to specify design functionality, and produce executable prototypes at an early stage in the design flow. The high level of abstraction provided by languages such as C is referred to as the *system level* or *electronic system level* (ESL) [KMN+00]. System level models can be used to experiment with design concepts, and to act as *gold standards* for testing against the RTL designs that are synthesized to hardware.

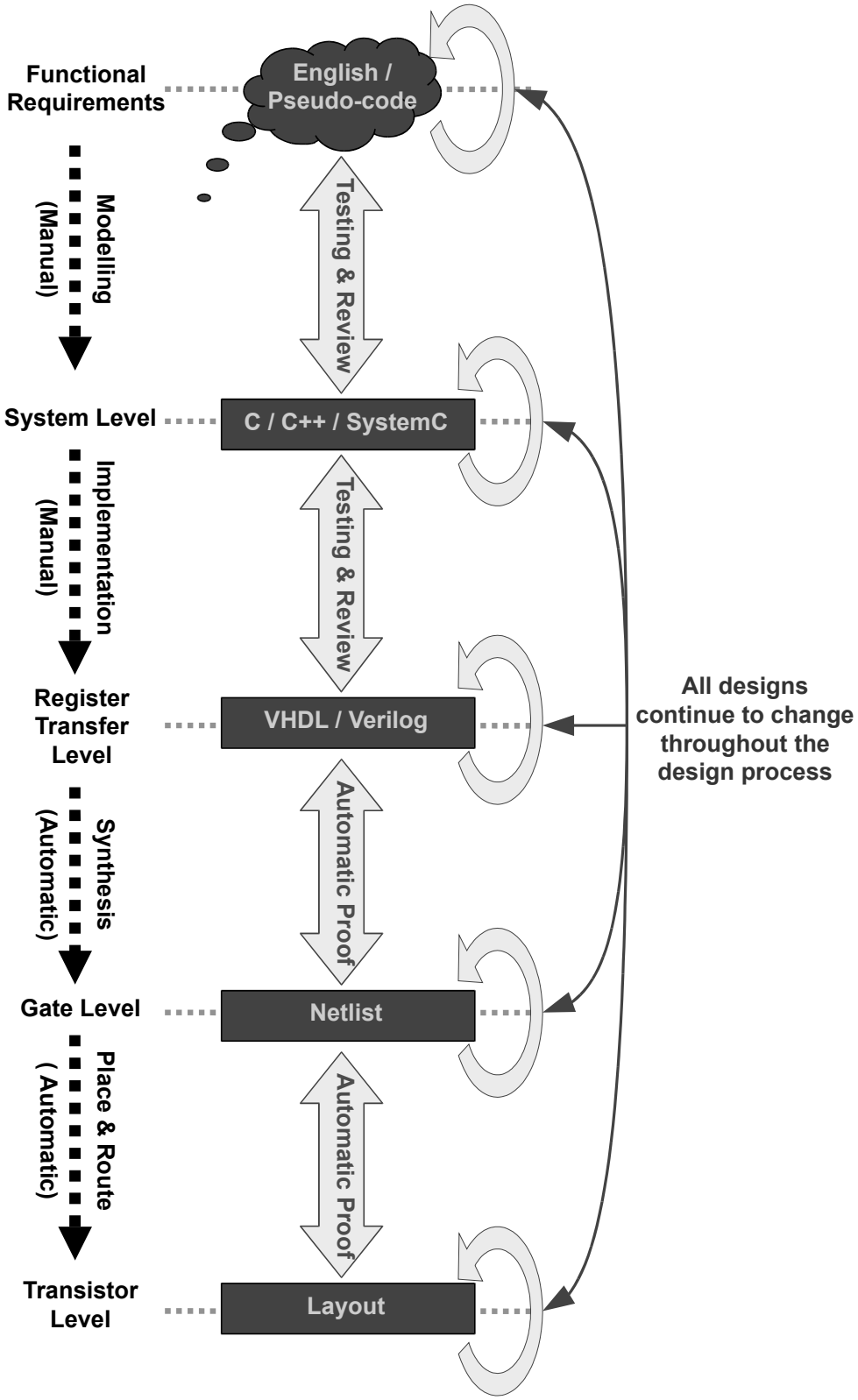

Figure 1.1: A Typical IC Design Flow, Including Functional Verification

1.2.1 The IC Design Flow

Figure 1.1 shows a modern IC design flow, by which simple initial concepts can be transformed into sophisticated IC designs with hundreds of millions of transistors. The design flow begins with a distillation of the design concept into a set of concrete functional requirements, which are typically expressed with a mixture of natural language prose, pseudo-code, tables, diagrams and so-on. This specification may be incomplete and ambiguous, and tends to mature as time passes. Next, an executable system level model is produced to animate the functional requirements at a high level of abstraction; this allows experimentation with the design concept, or *rapid prototyping*, which can stimulate modifications to the functional requirements. After this, a synthesizable RTL design is produced which implements the system level model in a way that attempts to meet any timing and power constraints. The development of RTL serves as a further exploration of the design concept, which can stimulate changes to the functional requirements and the system level model. From here, a gate level *netlist* is synthesised from the RTL, and a transistor level *layout* is then automatically compiled from the gate level design using a *place and route* tool. Finally, in a process called *spinout*, the layout is converted to a mask, which is used in a lithography process to fabricate the actual silicon chip. In order to ensure that the final chip satisfies its timing, power and area constraints, the RTL, netlist and layout can be repeatedly modified before spinout.

1.2.2 Functional Verification

Functional verification is the process of assuring that a transistor level layout is functionally equivalent to its specification of requirements. At present, this is primarily achieved using:

1. **Testing and design review** at the system and register transfer levels.
2. **Formal equivalence checking** at the gate and transistor levels.

Testing and design review are fundamentally limited because they cannot verify designs *rigorously* and *exhaustively*, meaning that bugs can – and do – pass through the verification process undetected:

- Testing involves simulating designs in software. Because of the complexity of modern IC designs, they are typically simulated for only a small fraction of the

possible inputs and system states. Furthermore, as design complexity increases, *test coverage* – the fraction inputs and system states that can feasibly be checked – generally decreases.

- In the design review process, designs are manually compared to specifications written in natural language prose, pseudo-code, tables, diagrams and so-on. However, natural language specifications are often incomplete and ambiguous, and human reasoning can make oversights and draw false conclusions.

Formal equivalence checking does not have these problems; it uses automated proof to establish functional equivalence rigorously for *all* possible inputs. Clearly, this would be preferable at the system and register transfer levels, but the application of automated proof at these levels poses significant challenges (which motivate the present work). Automated proof is currently a secondary activity at the system and register transfer levels; it is applied in restricted areas to provide enhanced error detection, but falls short of a complete solution for functional verification.

As long as we lack the means to achieve rigorous functional verification at the system and register transfer levels, bugs will pass undetected into silicon chips. In fact, they are surprisingly common in modern ICs. For example, the data-sheet for Intel's Core i7 processor lists 136 known errors [Int10]. To illustrate, the first three are:

1. MCi_Status overflow bit may be incorrectly set on a single instance of a DTLB error;

2. Debug exception flags DR6.B0-B3 may be incorrect for disabled breakpoints;

3. MONITOR or CLFLUSH on the local XAPIC's address space results in hang.

Bugs are currently a fact of life in chip production. Most of the time they are relatively benign, and can be 'worked around' by programmers, but they can also cause serious problems. In 1994 a bug was discovered in the floating point unit of Intel's Pentium processor, which caused Intel to replace the faulty chips on demand, with an eventual cost of almost $500 million, as well as untold reputational damage [CB00]. Bugs can also be costly when they are discovered before the product is released. Chips often have a marketing lifespan in the order of months before they are superseded by the next generation of ICs fielded by the competition, and bugs found late in the production cycle can delay fabrication, reducing the lifespan of the product or even

rendering it obsolete before it ever enters the marketplace. Worse yet, bugs have the potential to cause tragic consequences; chips are now used in safety critical systems such as nuclear reactors, passenger aircraft and car engines, and a failure in one of these could cause loss of life.

1.3 Formal Methods

In the battle against exploding design complexity, the hardware industry is increasingly supplementing established *ad hoc* design practices with more rigorous mathematical techniques. As we have seen, functional verification at the system level and register transfer level is currently a flawed process; testing and code review allow design errors to pass through the design process unchecked. Formal methods are less affected by the inherent flaws that blight testing and code review, but there are significant unsolved technical challenges which prevent them from being used pervasively at these levels, as they are at the gate and transistor levels. Instead they are applied piecemeal, to raise the quality of functional verification incrementally. There is a focused effort across industry and academia to overcome these problems and thus increase the impact of formal methods in the earlier stages of the design flow.

The term *formal methods* covers a broad range of approaches which apply mathematical techniques to the design of computational systems. These include verification by mathematical proof, design with semantically elegant languages and a host of so-called *semi-formal* approaches that use formal techniques to enhance existing informal design and verification practices. Our discussion of formal methods begins with the notion of verification by mathematical proof, which is a primary motivation of the field and also leads to an intuitive understanding of other concepts, including semantically elegant languages.

1.3.1 Verification by Mathematical Proof

Proof can be applied to programs written in any language for which a semantics can be precisely defined; for such languages, the behaviour of programs can be inferred before they are actually run, using mathematical reasoning. Intended properties of a given program can be expressed as theorems, and proof techniques can be used to establish their validity under all conceivable external environments. This presents a new approach to the functional verification of IC designs, which is potentially more

rigorous than design review and more exhaustive than testing. Proof can offer a more systematic inspection of design functionality than *ad hoc* design review and, in contrast to testing, can verify properties for all possible inputs to a system.

The format of theorems and proofs can vary. Theorems can be written in natural language and proven with written prose, in the style which is common in text books and academic papers. This kind of specification and proof can be viewed as a refinement of the informal design review process. Instead of specifying a system with natural language prose or pseudo code, the system's requirements are partitioned into a set of concise natural language theorems which can be proven with structured, rigorous reasoning, in place of the unstructured *ad hoc* reasoning which is used in design reviews. However, theorems and proofs which are formulated in natural language inherit the intrinsic problems that were associated with design review in §1.2.2; natural language is ambiguous, allowing imprecision and misinterpretation, and proofs written in prose typically appeal to human reason, which can draw false conclusions and make oversights. Moreover, proofs require human effort to construct, and owing to the complex nature of IC designs, this can become extremely time consuming.

Alternatively, natural language can be abandoned in favour of a *formal logic*, which is a mathematical system for performing precise and unambiguous reasoning. It will typically include:

1. A *specification language* for describing systems and asserting theorems about them: for example, *higher-order logic* provides a simple functional programming language, together with universal and existential quantifiers ($\forall$ and $\exists$) which allow theorems to be expressed about the properties of functions and variables.

2. A set of *inference rules* which are used to prove theorems written in the specification language.

Formal logic reduces mathematical proof to a process of *calculation*. Whereas natural language proofs appeal to human reason, their formal counterparts are composed purely from the repetitive application of inference rules, which for many logics are proven to be *sound*, meaning that they cannot possibly be used to establish the truth of a theorem which is actually false or *vice versa*.

Formal logic has a long history in the literature. For example, the epic *Principia Mathematica* of Whitehead and Russell (completed in 1913) was an attempt to

derive all mathematical truths within formal logic[3]. However, formal proof was impractical for most purposes until the advent of computers, which allowed proofs to be *mechanised*, giving rise to the field of *automated reasoning*[4].

1.3.2 Clean Abstractions and Semantic Elegance

In principle, proof can be applied to programs written in any language, as long as the semantics of the language can be formally specified. However, languages vary in terms of the ease with which formal semantics can be expressed, and the tractability of proofs which use these semantics. Unfortunately, as we shall see in later chapters, *ad hoc* languages such as VHDL, Verilog and SystemC perform poorly in both respects: they have convoluted semantics [Klo95, Gor95, BGG+92, MRH+01] which often necessitate contrived proof techniques. Spurred on by these challenges, the formal methods community has developed an innovative class of languages which are comparable to the popular *ad hoc* languages in terms of their utility for system design, but also have elegant semantics for the purpose of simplifying proof. The subtle attribute of semantic elegance has been achieved with a technique which is familiar from our discussion of the evolution of hardware languages – abstraction.

Abstraction is the art of describing systems in a way which hides unwanted detail whilst retaining the properties of interest. Hardware description languages use abstraction to express hardware designs without describing the nitty-gritty details of the gate and transistor levels. In a similar way, semantically elegant languages use abstraction to express computational systems without describing implementation details such as pointers, garbage collection and memory resources. This simplifies the semantics without reducing the class of computations that can be described. In the extreme, there are *calculi* such as the λ-calculus[5] [Chu40], a minuscule language which has just two primitive computational mechanisms (function parameterisation and function application) but is Turing complete. The semantics of the λ-calculus can be described with three simple rules; with these rules, one can evaluate any program written in the language. In order to prove a property about the evaluation of a λ-calculus program, one must only consider these three evaluation rules; in contrast, in order to prove properties of a C program, one would need to consider the behaviour of

[3] In the event, it actually covered a fragment of set theory, cardinal numbers, ordinal numbers and the reals.

[4] Excellent introductions to automated reasoning are provided in [Har09] (for automated deduction) and [CGP00] (for model checking).

[5] For a comprehensive introduction, see [Pie02].

pointers, garbage collection and so-on.

Alongside the minimalist calculi, fully-featured programming languages have emerged which adhere to the spirit of abstraction and semantic elegance, but are usable for everyday programming tasks. For example, functional programming [Bir98, OGS08, Pie02] is a paradigm that closely resembles the λ-calculus. It is arguably the most well known programming formalism and is used extensively throughout this thesis.

Are Semantically Elegant Languages Usable?

The promise of semantically elegant languages would by diminished if they were not competitive with *ad hoc* languages in terms of utility for everyday design; fortunately, there is a growing body of evidence to show that they are. Functional languages, for example, are well-suited to formal proof, but are gaining popularity in the programming community for other reasons:

- Their abstract nature makes functional programs *concise*, which increases design efficiency by making programs less time-consuming to construct;
- Powerful type systems (which are another form of abstraction) make programs inherently less error prone compared to the popular imperative languages such as C [FSNB09, OGS08]. It is a truism that functional programs "tend to just work" once they compile.

In the hardware domain too, semantically elegant languages are having an impact on design productivity. For example, the language Bluespec SystemVerilog [Nik04], which is considered at length in this dissertation, uses novel abstractions from computer science to describe hardware in a way that is substantially more concise than RTL, thus reducing line count and design time, whilst producing hardware that is competitive with hand-written RTL in terms of time and area for many applications [WNRD04, Nik04].

1.4 Synopsis

This thesis investigates the application of proof-based verification to hardware designs expressed in two semantically elegant languages, namely Concurrent Haskell and Bluespec SystemVerilog.

Chapter 2 employs Concurrent Haskell for the task of specifying and formally verifying a novel network-on-chip architecture. A behavioural specification is constructed for the SpiNNaker NoC, which can be executed and verified by hand-proof. A novel routing mechanism is verified by mathematical induction.

Chapter 3 reviews Bluespec SystemVerilog, and introduces BSV implementations of Peterson's algorithm and the control circuitry of a round-robin arbiter, which serve as running examples for the application of automated reasoning in the chapters that follow.

Chapter 4 introduces the concept of automated reasoning and surveys the literature for automated reasoning strategies of potential application to Bluespec SystemVerilog. It is found that several automated theorem provers support experimentation with a broad range of proof strategies, including model checking, automatic abstraction and the full spectrum of deductive reasoning. These strategies can be applied to BSV designs, if BSV can be embedded in the theorem prover's logic.

Chapter 5 embeds a non-trivial subset of BSV in the higher order logic of the PVS theorem prover. BSV is found to be naturally suited to expression in formal logic. Its use of guarded actions as an underlying model of concurrency makes it similar to a host of languages used for model checking, and its functionally-inspired language constructs can be translated into logic almost verbatim, with a novel application of monadic techniques. To demonstrate the proposed embedding strategy, the two BSV examples from chapter 3 are translated by hand into the PVS logic.

Chapter 6 applies automated reasoning to PVS specifications produced with the monadic embedding strategy of chapter 5. It is found that temporal logic theorems concerning these specifications can be verified with a combination of model checking and deductive reasoning. Proof strategies are written in the PVS strategy language to automate this process.

Chapter 7 investigates the verification of BSV designs with a stand-alone model checker. The monadic embedding strategy is used to translate BSV designs into the specification language of the SAL model checker. It is found that SAL permits monadic specifications, but fails to model check them efficiently, owing of their heavy use of higher order functions. However, an equivalent but more verbose *primitive* embedding strategy is found to be compatible with the SAL model checker, and a verification strategy is presented which combines SAL model checking over primitive specifications with deductive reasoning in PVS to establish the equivalence of primitive and monadic BSV embeddings.

Chapter 8 presents a review of the literature which is related to the embeddings of Bluespec SystemVerilog presented in earlier chapters. It is found that BSV presents unique challenges in the application of automated reasoning, being more complex than other guarded action languages, and exceptional amongst hardware languages in its choice of guarded actions as the underlying model of concurrency.

Chapter 9 draws conclusions, and presents topics for further work. For BSV in particular, a number of natural extensions to the present work are discussed, including the application of automatic abstraction and compositional reasoning to monadic PVS specifications. Furthermore, the overall direction of the present work is placed into the context of an expansive – and mostly unexplored – research space concerning automated reasoning for BSV.

1.5 Publications and Source Code

The research contributions of this dissertation have also been published in the following papers:

1. DOMINIC RICHARDS AND DAVID LESTER. A monadic approach to automated reasoning for Bluespec SystemVerilog. In *Innovations in Systems and Software Engineering: a NASA Journal*, 7(2):85-95, 2011.

2. DOMINIC RICHARDS AND DAVID LESTER. A prototype embedding of Bluespec SystemVerilog in the PVS theorem prover. In *Proceedings of the 2nd NASA Formal Methods Symposium (NFM)*, 2010.

3. DOMINIC RICHARDS AND DAVID LESTER. A prototype embedding of Bluespec SystemVerilog in the SAL model checker. In *Proceedings of the 8th International Workshop on Designing Correct Circuits (DCC)*, 2010.

4. DOMINIC RICHARDS AND DAVID LESTER. Concurrent functions: a system for the verification of networks-on-chip. In *Proceedings of the 3rd International Workshop on Hardware Design and Functional Languages (HFL)*, 2009.

5. DAVID LESTER AND DOMINIC RICHARDS. Specification of a network-on-chip. In *Proceedings of the 20th UK Asynchronous Forum (UK-ASYNC)*, 2008.

Accompanying source code can be found online [RL11].

Chapter 2

Verifying a Network-on-Chip

A network-on-chip architecture is specified and verified with a lightweight application of formal methods. The NoC features in the SpiNNaker many-core processor, which is being developed for efficient simulation of large scale neural networks. It supports a complex and unbounded toroidal network, which disregards established design conventions intended to ensure freedom from deadlock and livelock. A behavioural specification is produced in Concurrent Haskell, and a proof is presented which verifies a novel routing mechanism with mathematical induction.

2.1 Networks-on-Chip

As silicon feature sizes continue to shrink, it is becoming possible to fit entire systems onto a single chip, giving rise to the System-on-Chip (SoC) design paradigm. Such systems are typically composed of individually-designed intellectual property (IP) blocks such as processor cores, memory elements, graphics processing units, hardware accelerators and field-programmable gate arrays. Systems-on-Chip have complex communications requirements which can be provided by networks-on-chip [KJS+02] – scaled-down computer networks which allow IP blocks to communicate by routing data through a system of data links and connecting nodes. NoCs can transmit data either by creating direct communication links between IP blocks (an approach known as *circuit switching*) or by routing data packets (referred to as *packet switching*). It is with the latter that we are concerned.

Circuit switched NoCs can have various topologies depending on the specific structure and purpose of the SoC; a popular topology, for example, is the mesh structure shown in figure 2.1. A node in a packet-switched NoC typically consists of:

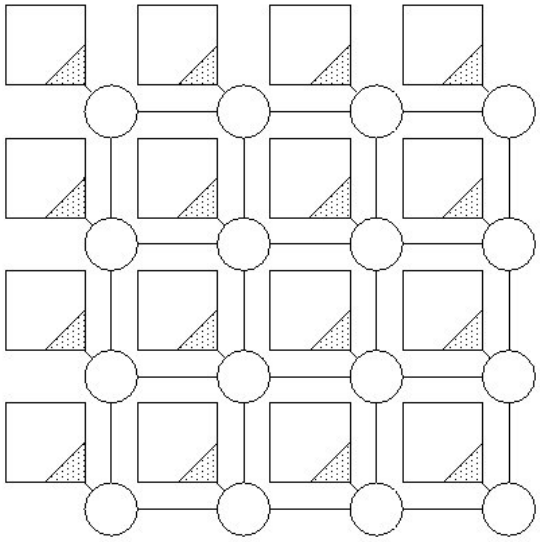

Figure 2.1: A Mesh NoC Topology

- **An arbiter** which merges the streams of packets received from a node's incoming data links into a single stream.
- **A router** which reads packets from the arbiter's output stream and forwards them along one or more of the node's outgoing data links.

NoC architectures can be highly complex and error prone, making them an attractive target for the application of formal methods. They have large state spaces and are often nondeterministic, which limits the effectiveness of traditional test-based verification techniques. Furthermore, being highly concurrent they are also prone to deadlock – a phenomenon that can stay hidden until late stage simulation or even post-fabrication testing.

2.2 SpiNNaker: A Spiking Neural Network Architecture

The SpiNNaker chip [FTB06b, FTB06a] is a novel many-core processor being developed at the University of Manchester in collaboration with the University of Southampton and ARM Ltd. It will support efficient real-time simulation of large-scale spiking neural networks. At the time of writing, the first batch of SpiNNaker chips is being fabricated.

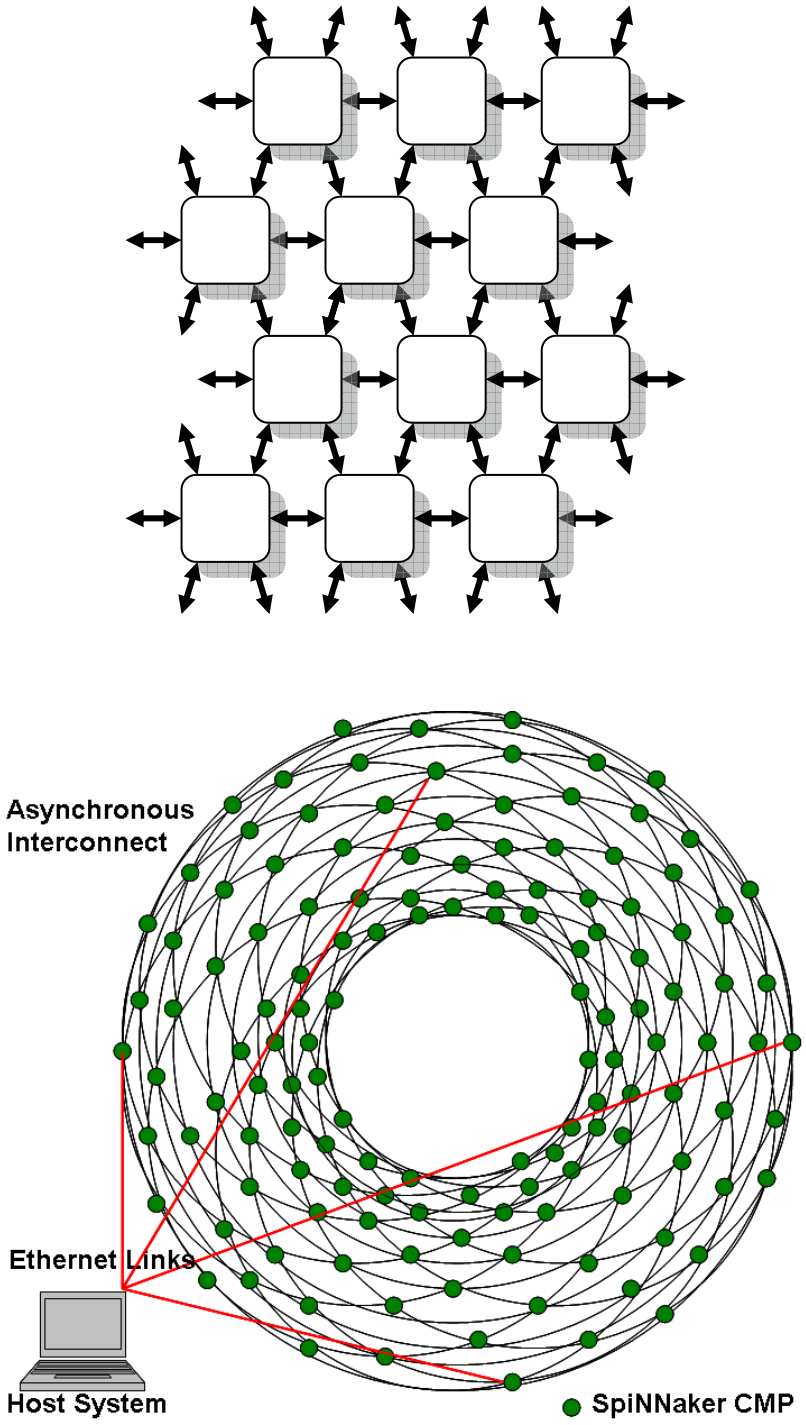

Figure 2.2: SpiNNaker system architecture. Top: each SpiNNaker chip is connected to its six nearest neighbours with bi-directional links. Bottom: the left and right sides of the mesh are connected, as are the top and bottom edges, to form a toroid. From [FT08].

The SpiNNaker chip (along with associated SDRAM chip) forms a node in a scalable parallel system, as shown in figure 2.2. On-chip processing power is provided by 18 ARM968 cores; together, these provide the integer processing power of a typical PC, but at much lower electrical power and in a compact physical form. The ARM cores are connected with a packet-switched network-on-chip, which is also connected to the NoCs of six neighbouring chips through inter-chip links, as shown in figure 2.3. Of the 18 ARM cores, 17 are designated to be *fascicle* processors which each model around 1,000 neurons in real time. The remaining core is designated as a *monitor* processor, which carries out operating system functions and provides the user with information concerning on-chip activity. A system-wide packet-switched communications network is formed by the concurrent operation of the individual networks-on-chip.

SpiNNaker systems will support neural simulations, in which neurons communicate by firing 'spikes' which are transmitted to other neurons. This is supported in the physical system by allowing neurons to launch source addressed packets into the system-wide communications fabric. When this happens, the NoCs distribute the packets across the system with *multicast, source addressed* routing. Each NoC has a content addressed memory unit (CAM), for which the packet's source address is used as a lookup key. The CAM of each NoC maintains a list of NoC outputs to which the packet should be forwarded. In this way, packets representing 'neural spikes' can be propagated throughout the system.

SpiNNaker chips adhere to the *Globally Asynchronous, Locally Synchronous* (GALS) design paradigm, in which each ARM core is individually clocked and connected to the rest of the chip by an asynchronous NoC [BF02]. Each chip, in turn, is connected to its six nearest neighbours by asynchronous inter-chip links.

Work is currently underway to produce systems with 500 and 50,000 nodes; the latter will provide approximately 200 teraIPS, which could simulate approximately 10^9 neurons, or 10% of the human cerebral cortex, using the initial target neural model.

2.2.1 Fault Tolerance and Emergency Routing

Owing to the vast size of planned SpiNNaker systems, component failure is a significant issue. Consequently, SpiNNaker incorporates two novel fault-tolerance mechanisms:

1. If a processor fails, its workload is migrated to other processors in real-time.

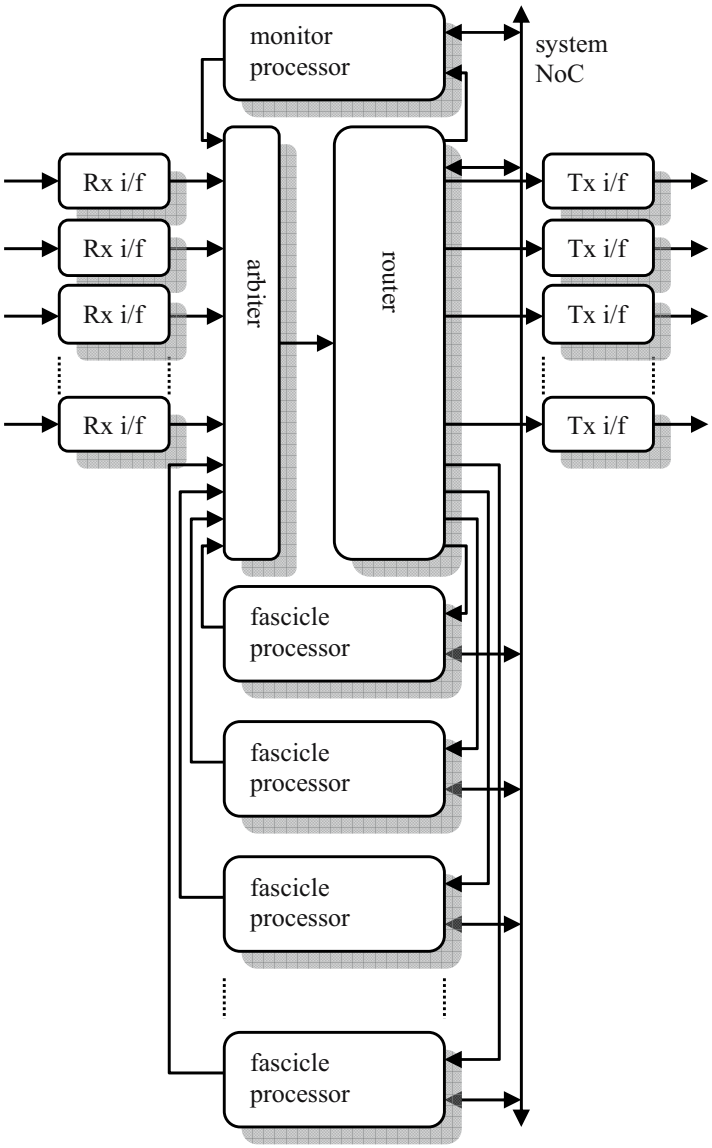

Figure 2.3: The internal structure of a SpiNNaker chip, illustrating the network-on-chip which is used to transmit 'spike' packets across the broader SpiNNaker system. Each fascicle processor models a cluster of neurons. Packets from neighbouring SpiNNaker chips arrive through the receiver interfaces (Rx i/f) and are merged with packets issued by the fascicle processors into a sequential stream by the arbiter. Each packet is then routed to one or more destinations, which may include neighbouring SpiNNaker chips (via the transmit interfaces Tx i/f) and/or local fascicle processors. The monitor processor carries out operating system functions and provides visibility to the user of on-chip activity. Processors are also connected to a separate 'system' NoC, which provides access to local memory, but is not discussed further in this thesis. From [FT08].

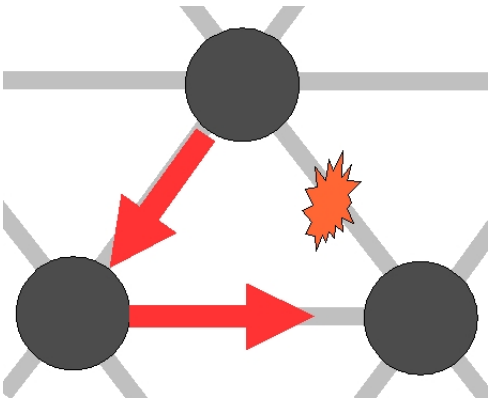

Figure 2.4: Emergency Routing.

2. If an inter-chip link fails (whether permanent or transient) traffic is handled in the first instance at the hardware level, by redirecting packets automatically via adjacent links, before invoking performance management software to carry out a more permanent solution.

The automatic re-directing of traffic around unresponsive inter-chip links is achieved by the concurrent operation of the surrounding chips' NoCs, as shown in figure 2.4. If an inter-chip link fails, packets are re-routed via a neighbouring chip. This mechanism is called emergency routing and proceeds in the following way.

Observe the fragment of a SpiNNaker system shown in figure 2.5. Imagine that chip A wants to send a packet down link AB to chip B, but link AB is broken. There are three possible cases:

1. The NoC router of chip A attaches a label "E1" (emergency routing, stage 1) to the packet and attempts to send it along the clockwise neighbour of the unresponsive link, which in this case is link AC. If link AC is broken, the packet is dropped; this unusual choice of behaviour is acceptable because of the fault tolerant nature of neural systems.

2. If link AC is functioning and chip C receives a packet with the label "E1" attached, it automatically changes the label to "E2" and attempts to send it along the clockwise neighbour of the link through which it entered the chip, which in this case is link CB. If link CB is broken, the packet is dropped.

3. If the packet reaches chip B, its NoC router removes the "E2" label and treats the packet as normal.

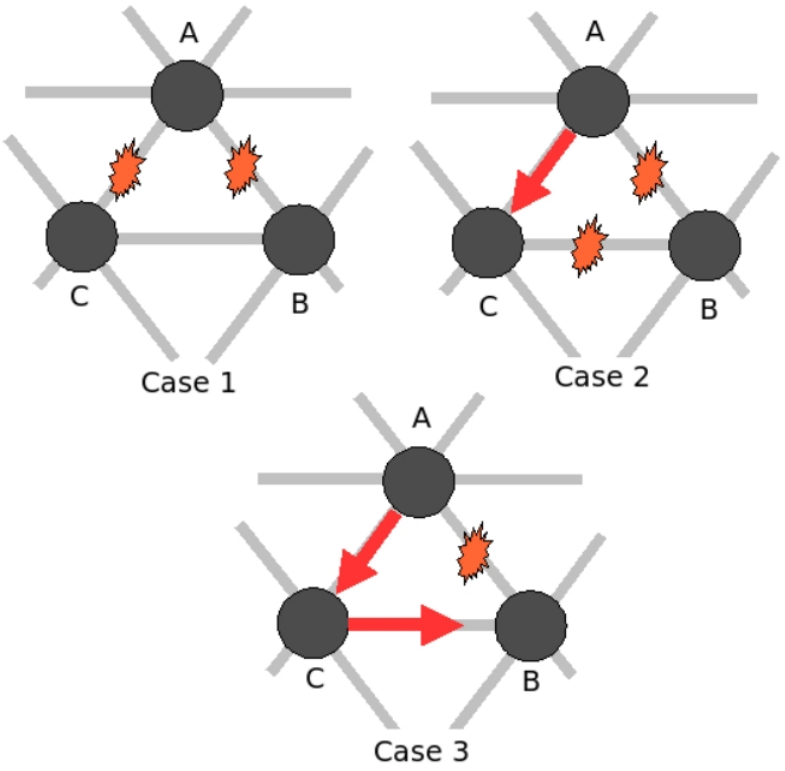

Figure 2.5: Emergency Routing Scenarios

If a packet is to be passed down an inter-chip link with the E1 label, it is possible that the same packet should be passed down this link anyway, with the N (normal) label. If this is the case, the router sends a single packet with type N_E1. When a packet of this type is received, it is split into an N packet and an E1 packet, which are then processed independently.

SpiNNaker NoCs also implement a 'default' routing mechanism, which allows packets to be routed through chips without entries being required in the chips' look-up tables. If a router receives a packet from an inter-chip link and the packet's source-address is not in the router's look-up table, it forwards the packet to the diametrically opposite inter-chip link. Because of this, look-up table entries are required only when packets either originate on the chip, are destined for on-chip processors, or change their 'direction of travel' at the chip.

P	$::=$	e	Haskell expression of type $IO()$
	$\mid$	$(P_1 \mid P_2)$	parallel composition
	$\mid$	$\langle\rangle_\alpha$	empty memory element named α
	$\mid$	$\langle x\rangle_\alpha$	memory element named α holding value x
	$\mid$	$(\nu\ \alpha)\ P$	restriction of the name 'α' to P
	$\mid$	$\emptyset$	the null process

Figure 2.6: A Grammar of Process Types for Concurrent Haskell

2.3 Concurrent Haskell

Concurrent Haskell [PJGF96] extends the lazy functional programming language Haskell with channel-based concurrency. Its concurrency primitives are accompanied by an elegant operational semantics, meaning that programs can serve as a basis for formal reasoning. This section introduces a subset of Concurrent Haskell. A comprehensive introduction to Haskell can be found in [Bir98].

Peyton-Jones *et al.* [PJGF96] present an operational semantics for Concurrent Haskell, which is based on the "chemical abstract machine" presentation of π-calculus [Mil93, Mil99]. The grammar of process types is given in figure 2.6. It tells us that the following are valid Concurrent Haskell processes:

1. e, whenever e is a Haskell statement of type $IO()$.

2. $P_1 \mid P_2$ – two processes executing concurrently.

3. $\langle\rangle_\alpha$ – an empty memory location with pointer α. (Notice that memory locations are defined as concurrent processes.)

4. $\langle x\rangle_\alpha$ – a full memory location.

5. $(\nu\ \alpha)\ P$, where α is a pointer for a memory location. This restricts the scope of α to P. The symbol ν is shorthand for 'new'; α is a *new* pointer for use in P.

Concurrent Haskell is implemented as a standard Haskell library which provides functions and types for creating concurrent processes and facilitating communication between them. We now discuss the following:

```
data MVar a

forkIO          :: IO () → IO ThreadId

newEmptyMVar    :: IO (MVar a)

putMVar         :: MVar a → a → IO ()

takeMVar        :: MVar a → IO a
```

Haskell

A value of type *MVar a* represents a pointer to a memory location that can hold one element of type α. This memory location can be used as a channel for communication between Concurrent Haskell processes. The semantics of the above functions are formally defined with a reaction relation, which is shown in figure 2.7. The notation $\mathscr{M}[f]$ can be read as "a monadic expression where the next function to be evaluated is f". For our purposes, this can be one of three things:

1. The monadic function f on its own; for example, $forkIO\ a$ for some Haskell function a.

2. A **do** statement of the form:

$$\begin{array}{l}\textbf{do}\\ \quad f\\ \quad\ < more\ statements >\end{array}$$

3. A **do** statement of the form:

$$\begin{array}{l}\textbf{do}\\ \quad x \leftarrow f\\ \quad\ < more\ statements >\end{array}$$

With this in mind, the above Concurrent Haskell functions are defined by figure 2.7 and their type declarations to have the following behaviour:

- The function $forkIO$ takes a Haskell expression of type $IO()$, generates a new thread for this expression to execute in, and returns a thread ID. Its behaviour is described formally by the rule $(Fork)$, which can be applied to any process of the form:

$$\mathscr{M}[forkIO\ a]$$

$$
\begin{array}{lcll}
(Fork) & \mathscr{M}[forkIO\ a] & \rightarrow & a \mid \mathscr{M}[return\ id] \\
 & & & \quad \text{where } id :: ThreadId \\
(New) & \mathscr{M}[newEmptyMVar] & \rightarrow & (\nu\ \alpha)(\langle\rangle_\alpha \mid \mathscr{M}[return\ \alpha]) \\
 & & & \quad \text{where } \alpha \notin free\_names(\mathscr{M}) \\
(Put) & \langle\rangle_\alpha \mid \mathscr{M}[putMVar\ \alpha\ x] & \rightarrow & \langle \mathtt{x} \rangle_\alpha \mid \mathscr{M}[return\ ()] \\
(Take) & \langle\ x\ \rangle_\alpha \mid \mathscr{M}[takeMVar\ \alpha] & \rightarrow & \langle\rangle_\alpha \mid \mathscr{M}[return\ x] \\
(Par) & P \mid Q & \rightarrow & P' \mid Q \quad if\ P \rightarrow P' \\
(Res) & (\nu\ \alpha)(P) & \rightarrow & (\nu\ \alpha)(P') \quad if\ P \rightarrow P'
\end{array}
$$

Figure 2.7: A Reaction Relation for Concurrent Haskell

and it would transform this process into a new process of the form:

$$a \mid \mathscr{M}[return\ id]$$

Here, a new process is created with the function 'a' being evaluated inside it.

- *newEmptyMVar* creates a new empty memory location and returns a pointer to it, wrapped in the IO Monad: this is specified by the rule (New).

- *putMVar* takes a pointer to a memory location and a value. If the memory location is empty, the rule (Put) dictates that *putMVar* will place the value into it. Note that there is no rule for the process:

$$\langle y \rangle_\alpha \mid \mathscr{M}[putMVar\ \alpha\ x]$$

  putMVar cannot write to a memory element which is already full. Hence, if it is called to write to a full location, it waits until the location becomes empty.

- *takeMVar* takes a pointer to a memory location. If the location is full its contents are removed, as specified by the rule $(Take)$, and wrapped in the IO Monad. There is no rule to read from an empty memory location, meaning that *takeMVar* will not execute until the location in question becomes full.

2.4 Specifying a Network-on-Chip

This section introduces a novel framework for the behavioural specification of packet-switched networks-on-chip using Concurrent Haskell. Generic abstractions are presented for key NoC components, which can be modified to capture the behaviour and topology of specific NoC architectures.

2.4.1 Packets and Physical Links

Packets are represented as instances of a Haskell datatype. The actual type is left undefined here, to be implemented for specific NoC architectures. In keeping with the behavioural (un-timed) level of abstraction, physical links are represented with unbounded 'first in, first out' (FIFO) buffers, which have the type *Chan* in Concurrent Haskell [Has11].

2.4.2 IP Blocks

Because we are concerned with the verification of NoCs, rather than the specific IP blocks they serve, IP blocks are specified as simple non-terminating processes which receive and generate NoC traffic:

```
mkIP :: Int → Chan Packet → Chan Packet → IO ThreadId
mkIP id input output
        = forkIO $ loop $ do
                              p ← readChan input
                              writeChan output (new_pack id p)
```

Haskell

This function ("make IP") initialises an IP block with a unique ID (*id*) and two *Chan*s (*input* and *output*) which serve as its interface to the NoC. The IP block exists in its own concurrent process, and continually reads packets from the NoC, creates new packets with the *new_pack* function, and launches them into the NoC to generate more network traffic. The *new_pack* function can be tailored to suit the individual NoC architecture.

2.4.3 Arbiters

Arbiters merge the incoming packet streams of a NoC node into a single stream which is input to the node's router. This behaviour can be expressed as a collection of

concurrent processes, one for each input stream, which read packets from the relevant input and write them to a common output. The following function creates an arbiter which merges packets, having first paired them with integers to indicate the input that each packet came from:

```
mkArbiter :: [Chan Packet] → Chan (Packet, Int) → IO [ThreadId]
mkArbiter inputs output = mapM listener $ zip inputs [0 .. ]
  where
    listener (input, id) = forkIO $ loop $ do
                                        p ← readChan input
                                        writeChan (p, id) output
```

Haskell

The function *mkArbiter* takes a list of *Chan*s representing the NoC node's inputs and another *Chan* representing the arbiter's output. It then calls the *listener* function for each of the NoC node's inputs: this is achieved using the standard Haskell function *mapM* which takes a function of type $a \to m\ b$ (for some monad m) and a list of type $[a]$ and applies the function to every element of the list, giving a new list of type $[m\ b]$, which it then converts to a value of type $m\ [b]$.

Arbiters must generally satisfy a *progress* requirement – any continuously available input value will always eventually be forwarded. The semantics of Concurrent Haskell (§2.3) makes no guarantees about this kind of progress. However, for the purposes of formal proof, we can show that certain theorems follow from an assumption of arbiter progress. This assumption would then need to be justified for the RTL implementation of the given arbiter.

2.4.4 Routers

Routers are specified as follows:

```
mkRouter :: (Packet → Maybe [Int])
            → Chan (Packet, Int) → [Chan Packet] → IO ThreadId
mkRouter lookup input outputs
    = forkIO $ loop $ do
                        (p, id) ← readChan input
                        route (lookup p) outputs id p
```

Haskell

This function takes the following arguments:

1. A lookup table which maps packets to values of type *Maybe* [*Int*] which can be either *Nothing*, representing a lookup table miss, of *Just xs*, representing a lookup table hit which provides a list of integers to indicate the NoC nodes to which the packet in question should be forwarded.

2. A *Chan* representing the router's input (which is also the arbiter's output).

3. A list of *Chan*s representing the inputs to the NoC nodes.

The functionality of *mkRouter* is to launch a new concurrent process which continually reads packets from the arbiter's output, consults the lookup table to obtain forwarding information and calls the auxiliary function *route* to execute the actual forwarding. The function *route* is left to be implemented for specific NoC architectures.

2.5 Specifying the SpiNNaker NoC

The toroidal structure of the system-wide communications network contains a myriad of cyclic paths, from small circuits between neighbouring nodes to Hamiltonian cycles along which the default routing mechanism will single-handedly carry packets through every node in the system. If any one of these paths is ever used, deadlock and/or livelock can ensue. Furthermore, two other novel features of SpiNNaker complicate the task of ensuring that they are not used:

1. Packets are routed through the system according to the contents of NoC CAMs, which are generated by third parties.

2. Emergency routing spontaneously alters the courses of packets.

Externally generated CAMs will be verified with software which checks for cyclic routing paths, and each packet also carries a time stamp, allowing it to be dropped if it remains in the system for too long. The emergency routing mechanism has been verified with testing, code review and the formal approach presented here.

2.5.1 Packets and Inter-Chip Links

In the SpiNNaker system, packets are routed according to the unique 'address' of the neuron which initiated them. However, we abstract away from the concept of neurons and assign unique IDs to the ARM cores; this simplifies the specification of ARM core

functionality, whilst still supporting the NoC's source-addressed routing functionality. Packets carry the ID of their originating core, and also a 'packet type' for emergency routing:

```
type Packet       = (Int, PacketType)
data PacketType = N | E1 | N_E1 | E2
```
Haskell

On-chip and inter-chip links are represented as unbounded FIFO buffers, which can be specified as *Working* or *Broken*:

```
data Channel = Working (Chan Packet) | Broken (Chan Packet)
```
Haskell

2.5.2 ARM Cores

ARM cores are specified as IP blocks from §2.4.2. The *new_pack* function is defined as follows:

```
new_pack :: Int → Packet → Packet
new_pack id (src, N) = (id, N)
```
Haskell

The behaviour of an ARM core is to receive packets form the NoC and issue new packets, carrying its own unique ID, back into the NoC.

2.5.3 NoC Arbiters and Routers

Arbiters and routers are specified with the approaches presented in §2.4.3 and §2.4.4 respectively. The function *route*, which was left undefined in §2.4.4, is given in figure 2.8 along with its auxiliary function *writeToICLs*; *route* takes slightly different arguments to its counterpart in §2.4.4, as explained below.

The *route* function is responsible for copying packets to the appropriate inter-chip links and on-chip ARM cores, as well as initiating and handling emergency routing, and detecting erroneous packets. It takes the following arguments:

- *dests* :: *Maybe*[*Int*] – the information returned by the lookup table. *Nothing* represents a lookup table miss; *Just ds* indicates the on-chip and inter-chip links

```
route :: Maybe [Int] → [Channel] → [Channel] → Int → Packet → IO()
route dests procIns iclIns src (id, state)
  | state == N_E1 = do
                      route dests procIns iclIns src (id, N)
                      route dests procIns iclIns src (id, E1)
  | localMiss     = localMissHandler
  | forwardPacket = forward iclIns src (id, state)
  | otherwise     = let Just ds = dests in
                    do
                      writeToICLs iclIns  (filter (< 6) ds) (id, state)
                      writeToProcs procIns (filter (≥ 6) ds) (id, state)
  where
    localMiss     = dests == Nothing ∧ src > 5
    forwardPacket = state == E1 ∨ dests == Nothing

writeToICLs :: [Channel] → [Int] → Packet → IO ()
writeToICLs iclIns dests pack = mapM_ (writeToICL pack) [0 .. 5]
  where
    writeToICL (id, N) node
      | normalPacket      node = writeToChan (iclIns!!node) (id, N )
      | emergencyPacket   node = writeToChan (iclIns!!node) (id, E1 )
      | normAndEmerg      node = writeToChan (iclIns!!node) (id, N_E1)
      | dropped           node = drop (id, N)
      | otherwise              = return ()
      where
        normalPacket    node =  (busy     node) ∧ (¬ (jammed (node + 1))
        emergencyPacket node =  (quiet    node) ∧ (   jammed (node + 1))
        normAndEmerg    node =  (busy     node) ∧ (   jammed (node + 1))
        dropped         node =  (jammed   node) ∧ (   broken (node − 1))
        quiet           node = ¬(receiving node) ∧ ¬ ( broken node)
        jammed          node =   receiving node  ∧   broken node
        busy            node =   receiving node  ∧ ¬ ( broken node)
```

Haskell

Figure 2.8: A component of the SpiNNaker NoC specification.

that the packet should be forwarded to. The six inter-chip links are indexed $[0..5]$ and the 18 ARM cores are indexed $[6..23]$.

- $procIns :: [Channel]$ – a list of channels used for communication with ARM cores.
- $iclIns :: [Channel]$ – a list of channels used as inter-chip links.
- $src :: Int$ – an integer representing the NoC node (ARM core or inter-chip link) from which the packet entered the NoC.
- $(id, state) :: Packet$ – the packet.

Note that $route$ takes separate *Channel* lists for on-chip and inter-chip links, whereas its counterpart in §2.4.4 took only one list containing all links; this small difference simplifies the definition of $route$ and its auxiliary functions.

The function $route$ evaluates as follows:

- If the packet has type N_E1, it is split into two packets. Both are identical to the original except that one has packet type N and one $E1$: $route$ is then called recursively on each of these.
- If the lookup table returns $Nothing$ and the packet came from an on-chip processor, the predicate $localMiss$ evaluates to true, indicating that an error has been detected. This situation should never happen, because the lookup table should only return $Nothing$ for default routing when a packet enters the chip through an inter-chip link, and is forwarded to the diametrically opposite inter-chip link. The function $localMissHandler$ is called.
- Whenever the state is not N or the lookup returns $Nothing$, $forwardPacket$ evaluates to true and we have one of two scenarios:
  1. If the state is not N, we have a packet which is being passed from an inter-chip link for emergency routing. (We know from the definition of new_pack that only N-type packets originate from the on-chip ARM cores.)
  2. If $dests == Nothing$, we know from the falsity of $localMiss$ that the packet came from an inter-chip link, and so we have a valid case for default routing.

The cases for default and emergency routing have in common the fact that they can be handled without referring to the lookup table. For this reason, they are both passed to a function $forward$ which deals with this scenario.

- *otherwise*: if none of the guards evaluate to true, we know that we have a packet with type N, for which the lookup returned $Just\ ds$. In this case, $writeToICLs$ is called to write to the appropriate inter-chip links, and $writeToProcs$ is called to write to the appropriate ARM cores.

The function $writeToProcs$ is trivial and has been omitted here. However, $writeToICLs$ performs an important role in emergency routing, and is shown in figure 2.8. When it is called to forward a packet, $writeToICLs$ iterates through the inter-chip links (using the $mapM\_$ function[1]) looking for the following conditions on each link:

- If the link is working but should not receive the packet, it may still have to carry an emergency routing packet. The anti-clockwise neighbour is inspected; if it is broken and should receive the packet, the emergency-routing procedure dictates that the packet should be sent down the present link. This condition is recognised by the predicate *emergencyPacket*.

- If the link is working and the packet should be passed down it, the anti-clockwise neighbour is inspected;

  1. If it is broken and should receive the packet, we have a case for emergency routing. An N_E1 packet should be sent down the present link. This condition is recognised by the predicate *normAndEmerg*.

  2. If it is working, or not due to receive a packet, an N packet is sent down the present link. This condition is recognised by the predicate *normalPacket*.

- If the link is broken, the anti-clockwise neighbour is inspected; if it is broken and should receive the packet, the emergency-routing procedure dictates that the packet should be dropped. This condition is recognised by the predicate $dropped$.

- If none of the above conditions have occurred, no action should be taken for the link in question.

[1] $mapM\_$ is similar to $mapM$, except that it returns an instance of $IO\ ()$.

2.6 Verifying the SpiNNaker NoC

It will now be shown that a packet passing through the SpiNNaker system-wide communications network will never enter a cyclic path as a result of emergency routing. When a packet is launched into the network, we define the *intended path* as the path which it will take in the absence of broken inter-chip links (henceforth *links*) and the *realised path* as the actual path it takes, which may deviate from the intended path because of broken links. To avoid livelock, intended routing paths are checked at compile time to ensure they are acyclic. Because SpiNNaker supports multicast routing, (acyclic) intended routing paths are trees. We define a *hop* as the traversal of a single link. Breaks are assumed to be constant: links which are broken at the creation of the system will remain so, and links which are functioning at the creation of the system will not break.

Lemma 1: *No realised path will deviate from its intended path for more than one hop.*

Proof: A realised path will only deviate from its intended path if a link on the intended path is broken. In this case, the packet will diverge for one hop with the type $E1$. When a SpiNNaker NoC receives an $E1$ packet, it will either return it to its intended path or drop it. (These scenarios are shown graphically in figure 2.5.) □

Theorem 1: *For an acyclic intended path, every realised path will also be acyclic.*

Proof: We proceed by mathematical induction on the number of broken links in the system.

Base Case: No broken links. The realised path will be identical to the intended path, which is acyclic.

Induction hypothesis: If the realised path is acyclic in a given system with n broken links (henceforth the (n)-system) then it will also be acyclic in a system which is identical, except that it contains one extra broken link (henceforth the $(n+1)$-system).

Inductive Step: We have an acyclic realised path through a system with n broken links, and we wish to show that insertion of new break anywhere in the system would not produce a cyclic realised path. From Lemma 1 and the earlier system description, we know that the realised path must follow the intended path except that:

- It may deviate from the intended path any number of times, although each

deviation will be for at most one hop;

- Any branch of the intended path may terminate early in the realised path, if the packet is dropped because of broken links;
- A packet will only be dropped by a chip which is on the intended path, or deviates from it by one hop.

We have the following cases:

1. If the new break is not on the realised path in the (n)-system, then the realised path in the $(n+1)$-system will be identical to the realised path in the (n)-system, which is acyclic.
2. If the new break is on the realised path, but not the intended path, of the (n)-system, then it must have been an $E1$ hop. If such a link is broken, the packet will be dropped, meaning that one branch of the realised path in the $(n+1)$-system will terminate immediately before the new break, but elsewhere it will be identical to the acyclic realised path in the (n)-system.
3. If the new break is on both the realised and intended paths of the (n)-system, then it is an N hop. We have two cases:
   (a) If both of the links required for emergency routing are functioning, the realised path in the $(n+1)$-system will be identical to the acyclic realised path in the (n)-system, except for the two new emergency routing hops which are also acyclic.
   (b) If either of the links required for emergency routing is broken, the packet will be dropped, meaning that one branch of the realised path in the $(n+1)$-system will terminate immediately before the new break, but elsewhere it will be identical to the acyclic realised path in the (n)-system. □

2.7 Related Work

The work presented in this chapter is novel in its application of Concurrent Haskell for the specification of NoCs, and also in its verification of an innovative routing mechanism belonging to the SpiNNaker NoC, which was achieved by mathematical induction.

A number of hand-based formalisms exist for specifying and formally verifying concurrent systems such as NoCs. Examples include the π-calculus [Mil93], TLA$^+$ [Lam02], UNITY [CM88] and Event-B [AH07]. When compared to these formalisms, Concurrent Haskell has the advantage of being fully executable, and also provides the expressivity of a mature functional language. In these respects, it is a convenient language for rapid-prototyping at the system-level and, as we have seen, is amenable to verification by deductive proof.

There have been several attempts to verify NoC architectures with mechanised deduction, all of which have used pure functional languages to form abstract specifications of the systems in question. Gebremichael *et al.* [GVZ+05] use the PVS theorem prover [ORS92] to establish the absence of a class of deadlock scenarios in the Æthereal NoC protocol [GDR05]. They achieved this by identifying cyclic resource dependencies for buffers in the NoC, and verifying that if such a cycle exists, one or more of the resources will be unused, thus preventing deadlock. The first general framework for the application of theorem proving to NoC verification was the GeNoC system [BHPS07, SB08], which used the ACL2 theorem prover to provide a generic functional description of NoC architectures. This generic specification can be tailored to specific architectures by providing details such as network topology and routing algorithms. It has been used to verify properties of the HERMES and Octagon architectures. Böhm and Melham [BM08] present a refinement-based approach for the design of on-chip communication protocols, in which protocols are specified in Isabelle/HOL and refinement steps are proven to maintain correctness. When compared to these approaches, the present mothodology provides less robust formal assurances, because proof is conducted by hand. However, specification with Concurrent Haskell has the following advantages:

1. Its expressive concurrency primitives allow direct behavioural specification of NoC architectures, whereas the input logics of theorem provers require more abstract formulations.

2. It is executable, meaning that formal specifications can also serve as rapid prototypes.

3. It is convenient. Models are straightforward to construct, and verification with hand-proof is comparatively painless. In contrast, specification and verification with an automated theorem prover requires a good deal more expertise, and time.

2.8 Summary

A novel network-on-chip architecture belonging to the SpiNNaker many-core processor has been specified in Concurrent Haskell, and an induction proof has been presented for the verification of a novel emergency routing protocol. Concurrent Haskell has been shown to provide a suitable basis for the application of lightweight formal methods to NoC verification. Owing to its clean semantics, it allows the construction of elegant behavioural specifications which can be executed (Concurrent Haskell is a mature software language) and also subjected to formal reasoning with hand proof. The work presented in this chapter was carried out as a component of the SpiNNaker design cycle, in response to verification concerns expressed by the design team. It was found that a lightweight application of formal methods stimulated a closer inspection of the relevant NoC design concepts, which provided timely and valuable reassurances for the design team.

Chapter 3

Bluespec SystemVerilog

A review is presented of the hardware language Bluespec SystemVerilog. BSV is used to design two hardware components, which implement Peterson's algorithm and an arbiter control circuit, and will serve as running examples in later chapters when automated reasoning techniques are developed for the language.

BSV is a hardware description language based on guarded atomic actions. It has been shown to reduce design time when compared to hand-written RTL whilst producing comparable hardware for many applications [Nik04, WNRD04]. It is an industrially-mature language, being owned and developed by Bluespec Inc. and used by companies including Intel, IBM, Qualcomm, Panasonic, Fujitsu and STMicroelectronics.

3.1 Syntax and Semantics

BSV expresses hardware in terms of guarded atomic actions. The basic structural unit of a BSV design is the *module*, which defines a state type, an initialisation of that state, and a number of *methods* and *rules*. Modules can be *instantiated* by creating a new instance of the state type, with all values set to the correct initial values (in the same way that a Java object is instantiated from a class, for example). Methods are side-effecting functions which are used to access the state of a module instance, and possibly transform it in the process (as with the methods of a Java object). Rules are guarded atomic actions which can *fire* spontaneously to change the state of a module instance. Each rule has:

1. A *guard* which is a predicate on the state.

2. An *action* which is a sequence of statements that can be formed from the methods provided by local instantiations of other modules.

A rule can fire if its guard evaluates to `True` for the current state. When a rule fires, the statements in the action transform the state. The state updates occur *atomically*; they are applied in unison, and no other rule can change the state while this happens.

A simple example of a module is `mkReg`, which 'makes' a register that implements the following interface:

```
interface Reg #(type a_type);
  method Action _write(a_type x1);
  method a_type _read();
endinterface: Reg
```
BSV

This interface has the type parameter `a_type` and declares two methods, `_read` and `_write`. Other modules can instantiate `mkReg` to create a register, and use its methods in their own rules and methods. For example, we can instantiate two `Bool` registers called `request` and `acknowledge`:

```
Reg#(Bool) request     <- mkReg(False);
Reg#(Bool) acknowledge <- mkReg(False);
```
BSV

We can then write a simple rule that initiates a handshake protocol using these two registers by checking to see whether they are both set to `False`, and when they are, changing the state of `request` to `True`:

```
rule request_rl (!(request._read() || acknowledge._read()));
  request._write(True);
endrule
```
BSV

BSV users will notice that method calls for registers do not use the standard syntactic sugar; this relatively trivial point simplifies the semantics for embedding in later chapters.

In general, a rule has the form:

```
rule my_rule (rl_guard);
  statement_1;
  statement_2;
  ...
endrule
```
BSV

When a rule fires, its statements are applied *in parallel*; each acts on the state as it was immediately before the rule fired, meaning that the changes made by `statement_1` are not seen by `statement_2`. To permit this parallel application, statements are not allowed to conflict with each other by writing to the same elements of state.

3.1.1 One-Rule-at-a-Time Semantics

The behaviour of module instances is defined by an exceedingly simple *one-rule-at-a-time* semantics; a module instance evolves from a given state by allowing any *one* rule to fire. If more than one rule is ready to fire, an *unspecified* choice is made; there is *no fairness condition* placed on this choice, so a rule is not guaranteed to fire if there are always competing rules. When hardware is generated, a number of optimisations are made (for example, a clock is introduced and multiple rules are executed per clock cycle) but the behaviour will always comply with the one-rule-at-a-time semantics.

The one-rule-at-a-time semantics provides a *binary transition relation* for module instances. As we shall see in chapter 4, these are used extensively in the formal methods community to specify state machines for automated proof. This makes BSV naturally amenable to verification with such tools. In contrast, the mainstream *ad hoc* languages such as VHDL, Verilog and SystemC have convoluted *simulation cycle semantics*, which are difficult to describe in formal logic [BGG+92, Gor95, Klo95, MRH+01].

3.1.2 Static Elaboration and Staging

BSV provides language constructs for static elaboration, which allow the definition of sets of module instances, interfaces, rules and so-on for expansion before synthesis. For example, the following expression creates several instance of `request_rl` to operate on different sets of registers which are held in two lists, `req` and `ack`:

```
Integer i; // a compile time variable

for ( i = 0 ; i < 10 ; i = i + 1 ) begin
  rule request_rl (!(req[i]._read() || ack[i]._read()));
    req[i]._write(True);
  endrule
end
```
BSV

There are two stages in processing a BSV design:

1. **The Static Stage** – static elaboration is performed by executing the *static semantics*;

2. **The Dynamic Stage** – the design is executed according to the *dynamic semantics*, which is guaranteed to be consistent with the one-rule-at-a-time semantics from § 3.1.1. The design is executed either in a simulator or as real hardware.

Later chapters will see the dynamic semantics embedded in several proof tools. However, the static semantics will not be considered further in this dissertation, except in relation to further work. It should be noted that perfectly legal BSV code can be written without the use of static constructs.

3.2 Peterson's Algorithm

This section and the one following it introduce BSV hardware designs which will serve as running examples in later chapters, as automated reasoning techniques are developed for the language. We begin with Peterson's algorithm [Pet81] – a protocol which allows two processes to share a common resource without conflict, as defined by the following properties:

1. **Safety** – at most one process should have access to the shared resource at a given time.

2. **Progress** – a process which attempts to access the shared resource should always eventually succeed.

In addition to these properties, an implementation of Peterson's algorithm should not be able to enter a *deadlock state* from which it cannot progress. If this were to happen, the above properties might become true by default, although the system would no longer be functioning.

The two processes coordinate their actions by inspecting and modifying three elements of state: two *program counters* (one for each process) and a boolean *turn* flag. The program counters can be in one of three states: *sleeping*, *trying*, or *critical*. A process begins life in the sleeping state, and thereafter has the following behaviour:

1. A *sleeping* process remains inactive for an undetermined (and possibly infinite) period of time, until it wakes up and enters the *trying* state.

2. A *trying* process attempts to gain access to the resource continuously until it succeeds, whereupon it enters the *critical* state.

3. A *critical* process has gained access to the resource. It retains access for an undetermined (but finite) amount of time and then relinquishes it to return to the *sleeping* state.

Safety is achieved by ensuring that neither process becomes *critical* while the other is already *critical*. Progress is ensured by manipulating the *turn* flag to ensure that the processes are alternately given priority in the case of a resource-access conflict.

We now implement Peterson's algorithm as a BSV module; in later chapters, this module will be translated into logic and verified with respect to the above safety and progress properties using automated reasoning. We begin by defining an enumerated type for the program counters:

```
typedef enum {Sleeping, Trying, Critical} PC
  deriving (Bits, Eq);
```
BSV

The `deriving` clause instructs the compiler to produce a bit-level representation and test for equality. We now create two instances of the `Reg` module to hold the program counters:

```
Reg#(PC) pcp <- mkReg(Sleeping);
Reg#(PC) pcq <- mkReg(Sleeping);
```
BSV

We call our two processes p and q. We also create two more module instances, namely a Boolean register to hold the *turn* flag and a single element Boolean FIFO register which represents the shared resource:

```
Reg#(Bool)   turn <- mkReg(True);
FIFO1#(Bool) fifo <- mkFIFO1;
```
BSV

The FIFO will be accessed by processes which have entered the `Critical` state – it is not strictly necessary for a formulation of Peterson's algorithm, but is added to create extra complexity for the embeddings presented in later chapters.

We now create rules to define the behaviour of the two processes:

1. If a process is `Sleeping`, it enters the `Trying` state and sets the `turn` register to give the other process priority. (Process p has priority when the `turn` register holds the value `True`, and process q has priority otherwise.) For p and q respectively:

```
rule wake_p (pcp._read == Sleeping);
  pcp._write (Trying);
  turn._write (False);
endrule

rule wake_q (pcq._read == Sleeping);
  pcq._write (Trying);
  turn._write (True);
endrule
```
BSV

2. If a process is `Trying`, and either has priority or the other process is `Sleeping`, it can enter the `Critical` state:

```
rule grant_p (pcp._read == Trying
              && (turn._read || pcq._read == Sleeping));
  pcp._write (Critical);
endrule
```
BSV

```
rule grant_q (pcq._read == Trying
              && (!turn._read || pcp._read == Sleeping));
  pcq._write (Critical);
endrule
```
BSV

3. If a process is `Critical` and `fifo` is not full, it writes an element of data to `fifo` and returns to the `Sleeping` state, whilst also setting `turn` to give the other process priority:

```
rule p_critical (pcp._read == Critical);
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule

rule q_critical (pcq._read == Critical);
  fifo.enq (False);
  pcq._write (Sleeping);
  turn._write (True);
endrule
```
BSV

   The FIFO `_write` method carries an *implicit condition* which permits it to be called only when the FIFO is not full. Any rule which uses this method can only fire when the implicit condition is satisfied.

4. We also have a rule to empty the FIFO whenever an element is placed into it:

```
rule read_fifo;
  fifo.deq;
endrule
```
BSV

   The FIFO `deq` method carries an implicit condition which permits it to be called only when the FIFO is not empty.

3.3 Arbiter Control Circuit

We now consider a BSV design for the control circuit of a 3-input arbiter, which should have the following properties:

1. **Safety** – at most one input should have access to the output at a given time.
2. **Progress** – an input which requests access to the output should always eventually gain it, as long as no input retains the output indefinitely.
3. **Deadlock freedom** – the arbiter should never reach a state from which it cannot progress, as long as its inputs are active.

Our arbiter control circuit will implement the following interface:

```
interface Arbiter;
  method Action request1 ();
  method Action request2 ();
  method Action request3 ();
  method Action relinquish1 ();
  method Action relinquish2 ();
  method Action relinquish3 ();
endinterface
```
BSV

An input will be able to request control of the output when it does not have it, and relinquish the output when it does. Note there are no methods to remove a request which has been placed but not yet granted.

We define a module to implement the above interface:

```
module mkArbiter (Arbiter);
  ...
endmodule
```
BSV

Within this module, each input is assigned a Boolean ‘request’ register, which is used to record requests when they are made. The arbiter also has a Boolean ‘acknowledge’ register for each input, which is used to record the fact that a request made by the relevant input has been acknowledged, and that it has exclusive access to the output.

```
Reg#(Bool) req1 <- mkReg(False);
Reg#(Bool) req2 <- mkReg(False);
Reg#(Bool) req3 <- mkReg(False);
Reg#(Bool) ack1 <- mkReg(False);
Reg#(Bool) ack2 <- mkReg(False);
Reg#(Bool) ack3 <- mkReg(False);
```
BSV

Request methods have the following form:

```
method Action request1 () if (!(req1._read || ack1._read));
  req1._write (True);
endmethod
```
BSV

The 'if' statement on the first line is an implicit condition which prevents any rule that uses this method from firing when either req1 or ack1 hold the value True. Methods to relinquish the output have the following form:

```
method Action relinquish1 () if (ack1._read && req1._read);
  req1._write (False);
endmethod
```
BSV

In order to guarantee progress, the arbiter also has a separate Boolean 'token' register for each input. Of the three 'token' registers, only one is set to True at any time. When an input 'has the token', the arbiter gives it priority over all other incoming requests. Each time a request is granted, the token is cycled round to another input.

```
Reg#(Bool) tok1 <- mkReg(True);
Reg#(Bool) tok2 <- mkReg(False);
Reg#(Bool) tok3 <- mkReg(False);
```
BSV

These registers are accompanied by a local *action* to permute the token (remember that the statements will be executed in parallel):

```
Action move_token = (action tok1 <= tok3;
                            tok2 <= tok1;
                            tok3 <= tok2;
                     endaction);
```
BSV

We create rules to say that an input's request can be acknowledged if it holds the token and the output is not currently occupied:

```
rule ack1_with_tok (tok1 && req1 && !(ack1 || ack2 || ack3));
  ack1._write (True);
  move_token;
endrule
```
BSV

When an input's request is granted, the token is permuted with `move_token`. We now create rules to say that an input's request can be acknowledged whenever the output is not currently occupied, regardless of the token position, and constrain these rules to fire only when the above rules do not, by use of a *scheduling attribute*:

```
rule ack1_without_tok (req1 && !(ack1 || ack2 || ack3));
  move_token;
  ack1._write (True);
endrule

(* descending_urgency
     = "ack1_with_tok, ack2_with_tok, ack3_with_tok,
        ack1_without_tok, ack2_without_tok, ack3_without_tok" *)
```
BSV

We also create rules to lower the acknowledge flag for an input when it no longer requires access to the output:

```
rule arbiter1_hs (ack1 && !req1);
  ack1._write (False);
endrule
```
BSV

```
import Arbiter::*; // Package containing Arbiter and mkArbiter

module mkTbArbiter (Empty);

  Arbiter arb <- mkArbiter();

  //-- Client Requests ---------------------------------------

  rule client1_req;
    arb.request1();
  endrule

  rule client2_req;
    arb.request2();
  endrule

  rule client3_req;
    arb.request3();
  endrule

  //-- Client-Side Handshakes --------------------------------

  rule client1_hs;
    arb.relinquish1();
  endrule

  rule client2_hs;
    arb.relinquish2();
  endrule

  rule client3_hs;
    arb.relinquish3();
  endrule

endmodule
```

BSV

Figure 3.1: A Test Bench for mkArbiter

This concludes the description of `mkArbiter`. Figure 3.1 presents a test bench that creates an instance of `mkArbiter` and invokes its methods. For each input, `mkTbArbiter` provides a rule which calls the relevant request method whenever possible (i.e. whenever the implicit condition for the method is satisfied, which occurs when the relevant `req` and `ack` flags are low). According to the one-rule-at-a-time semantics of BSV (§3.1.1) the system will always execute a rule when there are rules to be executed; it turns out that whenever all of the request and acknowledge flags are low, one of the `client_req` rules will fire. This will be proven in later chapters.

We also create a rule for each input to relinquish the output once it has been granted. It will be proven in later chapters that a `client_hs` rule will always eventually fire once access to the output has been granted.

It should be noted that the above system could be described in a more succinct (and unbounded) form with static elaboration, which is not discussed here because it will not be supported in the theorem prover and model checker embeddings presented in later chapters. It does seem possible to represent static elaboration with the embedding approach that will be presented, but this is left as a topic for further work.

3.4 Summary

A review has been conducted of the semantically elegant hardware language Bluespec SystemVerilog. Two examples have been presented, which implement Peterson's algorithm and the control circuit of an arbiter . They will serve as running examples in later chapters as automated reasoning techniques are developed for BSV, and are well suited to this investigation for two reasons:

1. They demonstrate a number of advanced language features: module definition and instantiation, side-effecting methods with and without implicit conditions, rule composition using methods from instantiated modules, scheduling attributes.

2. They exhibit interesting and well-defined temporal properties, which can be specified in logic and verified with automated proof.

Chapter 4

Automated Reasoning for Bluespec SystemVerilog

The concept of automated reasoning is introduced, and the literature is surveyed for automated reasoning strategies of potential application to Bluespec SystemVerilog.

4.1 Automated Reasoning

Reasoning is part of the creative brilliance that sets us apart as humans, namely the ability to assimilate information from our surroundings, and use it to discover truths that would not otherwise be apparent. It allows us to undertake abstract endeavours such as the construction of complex computational systems, which we achieve by manipulation of semiconducting materials. It is an astonishing virtue, but also a fallible one; we can err and draw false conclusions, unaware of our incomplete knowledge, oversight, miscomprehension or miscalculation. In hardware design, for example, highly qualified verification engineers can pore over a design and meticulously, but erroneously, arrive at the conclusion of its correctness. Thus, the towering achievements of our reason can be undone by its fallibility. In our creativity then, can we overcome the fallibility of our own reason? Can we reduce reasoning to *calculation*?

The challenge of reducing reasoning to calculation has been studied for millennia in the field of logic. The Greek word *logos*, which was used by Plato and Aristotle to mean reason, can also mean computation. The link between reason and computation was made concrete by Gottfried Wilhelm von Leibniz (1664-1716), who suggested that a framework for systematic reasoning could be constructed with two components:

1. A *universal language* in which knowledge can be captured, and assertions about that knowledge can be expressed;

2. A *calculus of reasoning* which can be used to established the truth or falsity of assertions written in the universal language.

Leibniz suggested that such a system could be used to resolve disputes. He imagined a time when disagreements would culminate with a proclamation of *"calculamus!"* (let us calculate) whereupon the relevant background information and the point of contention would be translated into the universal language, and from this the truth would be computed. Logic has fallen short of this lofty aspiration! Remarkably, however, his approach is followed almost to the letter in the more restricted sphere of automated reasoning...

4.2 Logics and Decidability

Leibniz was remarkably accurate in his prediction of logical reasoning, but its modern incarnation differs on one point – rather than having a single universal language, we have a collection of logics, any of which may be preferable for a given reasoning task. Whatever its nature, however, a formal logic will conform to Leibniz's model, having a *specification language* for describing systems and asserting theorems about them and a set of *inference rules* which are used to prove theorems written in the specification language. Furthermore, we have the additional nuance that logics can be *unsound* (theorems can be proven which are, in fact, false), although this can be avoided by introducing a *type system*: from this point onwards, all logics are assumed to be typed.

Logics vary in terms of the properties they can express (their *expressivity*) and the ease with which proofs of those properties can be automated. In terms of automation, we have the following distinction:

- For some logics, a single *decision procedure* can be written to automatically prove or disprove any theorem; we call such logics *decidable*.

- For other logics, a *semi-decision procedure* can be written to prove any true theorem, although it may fail to terminate for theorems which are false; we call such logics *semi-decidable*.

- Other logics are *undecidable*, meaning that some degree of human innovation is required to construct proofs, which can either be encoded into a proof procedure

$$
\begin{array}{rcll}
p & ::= & x & \text{(boolean variable)} \\
 & | & true & \\
 & | & false & \\
 & | & \neg p_1 & \\
 & | & p_1 \wedge p_2 & \\
 & | & p_1 \vee p_2 & \\
 & | & p_1 \Rightarrow p_2 & \\
 & | & p_1 \Leftrightarrow p_2 &
\end{array}
$$

Figure 4.1: A Grammar of Propositional Logic

before the proof is attempted, or provided during the proof by human interaction.

As we shall see, expressivity and automation are often inversely related, creating an important design tradeoff when automated proof is considered for a given verification task: expressivity can be sacrificed for ease of automation, or a more expressive logic can be chosen at the cost of increased human input into the proof process. Decidability is clearly an attractive property when it comes to automated reasoning. However, high levels of automation can often be achieved for subsets of undecidable logics. Applications will often use a *fragment* of the logic in question. Typically, some fragments of an undecidable logic will be decidable, whilst other fragments will be semi-decidable, and customised *proof strategies* can be constructed for undecidable fragments, which use heuristics to assemble proofs, either autonomously or in combination with human guidance.

A extensive introduction to logic and automated reasoning can be found in John Harrison's engaging book [Har09][1]. This dissertation will feature propositional, first order and higher order logic, as well as several temporal logics.

4.3 Propositional, First Order and Higher Order Logic

Propositional, first order and higher order logic are closely related. Propositional logic is a subset of first order logic, which is a subset of higher order logic; consequently, propositional logic is less expressive than first order logic, which is less expressive than higher order logic.

[1] From which the above historical notes were taken.

$$
\begin{array}{rcl}
p & ::= & t \\
 & | & true \\
 & | & false \\
 & | & \neg p_1 \\
 & | & p_1 \wedge p_2 \\
 & | & p_1 \vee p_2 \\
 & | & p_1 \Rightarrow p_2 \\
 & | & p_1 \Leftrightarrow p_2 \\
 & | & \forall x.p_1 \\
 & | & \exists x.p_1 \\
 & & \\
t & ::= & x \\
 & | & f(t_1,\ldots,t_n), \text{ for } n \geq 0
\end{array}
$$

Figure 4.2: A Grammar of First Order Logic

- **Propositional Logic** is shown in Fig 4.1. It contains the boolean literals (*true* and *false*), boolean variables, negation, conjunction, disjunction and implication. Propositional logic is decidable.

- **First Order Logic** is shown in Fig 4.2. It extends propositional logic by replacing the boolean values with predicates on expressions written in a *functional specification language*, and also allows universal and existential quantification ($\forall$ and $\exists$) over the input values of functions. First order logic is undecidable, although some fragments are decidable and others are semi-decidable.

- **Higher Order Logic** extends first order logic by allowing quantification over functions, rather than simply the input values of functions. As with first order logic, higher order logic is undecidable, although some fragments are decidable and others are semi-decidable.

4.4 Temporal Logics

Temporal logics allow the formulation and proof of theorems concerning the evolution of state-based systems. For example, we can prove that from a given initial state,

a given proposition will hold in all future states, or will become true within a bounded or unbounded number of state transitions. An important class of temporal logic theorems is decidable, and many others theorems can be efficiently reduced to decidable theorems with the use of techniques such as abstraction. This section provides a brief introduction to temporal logic that will suffice for the present work: alternatively, a comprehensive introduction can be found in [CGP00].

4.4.1 Kripke Structures and Computation Trees

A nondeterministic state machine can be formally specified with a *Kripke structure*. A Kripke structure M over a set of *atomic propositions* AP is a 4-tuple (S, S_0, R, L) where:

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a nondeterministic transition relation on the state type.
- $L : S \to 2^{AP}$ is a function that labels each state with a set of atomic propositions which are true in that state. Atomic propositions are primitive truths about the state.

For model checking, a general restriction is that:

Transition relations must be left-total on S

That is to say, for a given transition relation $R \subseteq S \times S$, for every state $s \in S$, there must exist at least one $s' \in S$ such that $R(s, s')$. Model checking algorithms can give unpredictable results if they are applied to transition relations which violate this condition [CGP00]. Some model checking algorithms require the less restrictive condition that R must be left-total only for the reachable states (which are defined as the transitive closure of R on the initial states S_0).

We can think of the evolution of a nondeterministic state machine as a *computation tree*, which is a branching sequence of states formed by starting from an initial state and applying the transition relation recursively to give sequences of states. Because transition relations can be nondeterministic, a given state can map to more than one next state, creating branches in the computation tree.

$$
\begin{array}{rcl}
p & ::= & AP \\
 & | & \neg p_1 \\
 & | & p_1 \wedge p_2 \\
 & | & p_1 \vee p_2 \\
 & | & p_1 \Rightarrow p_2 \\
 & | & p_1 \Leftrightarrow p_2 \\
 & | & \mathbf{AX}\ p_1 \\
 & | & \mathbf{EX}\ p_1 \\
 & | & \mathbf{AF}\ p_1 \\
 & | & \mathbf{EF}\ p_1 \\
 & | & \mathbf{AG}\ p_1 \\
 & | & \mathbf{EG}\ p_1 \\
 & | & \mathbf{A}\ [p_1\ \mathbf{U}\ p_2] \\
 & | & \mathbf{E}\ [p_1\ \mathbf{U}\ p_2] \\
 & | & \mathbf{A}\ [p_1\ \mathbf{R}\ p_2] \\
 & | & \mathbf{E}\ [p_1\ \mathbf{R}\ p_2]
\end{array}
$$

Figure 4.3: A Grammar of Computation Tree Logic

4.4.2 Computation Tree Logic

A temporal logic theorem has the form:

$$M, s \models p$$

Here, p is a *temporal logic formula*, written in Computation Tree Logic (CTL), or another temporal logic. This theorem asserts that p is true for the Kripke structure M in state s.

Fig. 4.3 presents a grammar for CTL formulas. We have atomic propositions, negation, conjunction, disjunction, implication and eight *path quantifiers*. The path quantifiers have the following meanings:

- $M, s \models \mathbf{AX}\ p_1$ means that p_1 holds in all next states from state s.
- $M, s \models \mathbf{EX}\ p_1$ means that p_1 holds in at least one next state from state s.
- $M, s \models \mathbf{AF}\ p_1$ means that p_1 holds at some point along all paths from state s.

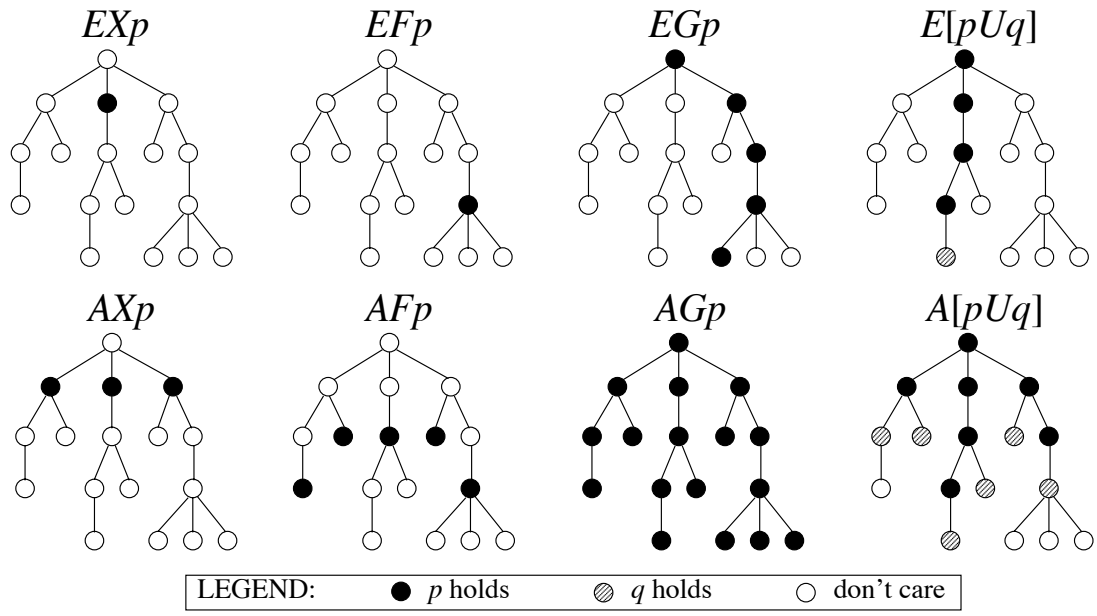

Figure 4.4: Computation Tree Logic (from [Sim07])

- $M, s \models \mathbf{EF}\ p_1$ means that p_1 holds at some point along at least one path from state s.
- $M, s \models \mathbf{AG}\ p_1$ means that p_1 holds continuously along all paths from state s.
- $M, s \models \mathbf{EG}\ p_1$ means that p_1 holds continuously along at least one path from state s.
- $M, s \models \mathbf{A}\ [p_1\ \mathbf{U}\ p_2]$ means that p_1 holds continuously until p_2 holds, along all paths from state s.
- $M, s \models \mathbf{E}\ [p_1\ \mathbf{U}\ p_2]$ means that p_1 holds continuously until p_2 holds, along at least one path from state s.
- $M, s \models \mathbf{A}\ [p_1\ \mathbf{R}\ p_2]$ means that p_1 holds continuously up to and including the first state in which p_2 holds, along all paths from state s.
- $M, s \models \mathbf{E}\ [p_1\ \mathbf{R}\ p_2]$ means that p_1 holds continuously up to and including the first state in which p_2 holds, along at least one path from state s.

Fig. 4.4 provides some illustrative examples of CTL formulas. To clarify their meaning, we discuss the top left example, which demonstrates a computation tree for which the theorem EXp is true for a given state (there exists a next state for which p

$$
\begin{array}{rcl}
p & ::= & AP \\
 & | & \neg p_1 \\
 & | & p_1 \wedge p_2 \\
 & | & p_1 \vee p_2 \\
 & | & p_1 \vee p_2 \\
 & | & p_1 \Rightarrow p_2 \\
 & | & p_1 \Leftrightarrow p_2 \\
 & | & \mathbf{X}\ p_1 \\
 & | & \mathbf{F}\ p_1 \\
 & | & \mathbf{G}\ p_1 \\
 & | & p_1\ \mathbf{U}\ p_2 \\
 & | & p_1\ \mathbf{R}\ p_2
\end{array}
$$

Figure 4.5: A Grammar of Linear Temporal Logic

holds). The state in question is represented by the root node in the tree, which has three branches to represent the transitions that are permitted by the transition relation. One of these transitions leads to a state in which p is satisfied, as represented by the black node.

4.4.3 Linear Temporal Logic

Linear Temporal Logic (LTL) is equivalent to the subset of CTL that excludes existential path quantifiers (EX, EF, EG, EU and ER). Because we only have universal quantifiers, we drop the A, and use simply X, F, G, U and R. Fig. 4.5 presents a grammar for LTL, and Fig. 4.6 provides illustrative examples.

4.5 Automatic Proof Tools

A variety of proof tools have been developed to automate reasoning in the different logics:

- For **propositional logic** we have *satisfiability (SAT) solvers* [GN02] and *satisfiability modulo theories (SMT) solvers* [DM06, DMB08], which provide fully automated solution of the boolean satisfiability problem. Many important problems reduce to satisfiability problems, including model checking [BCC+99],

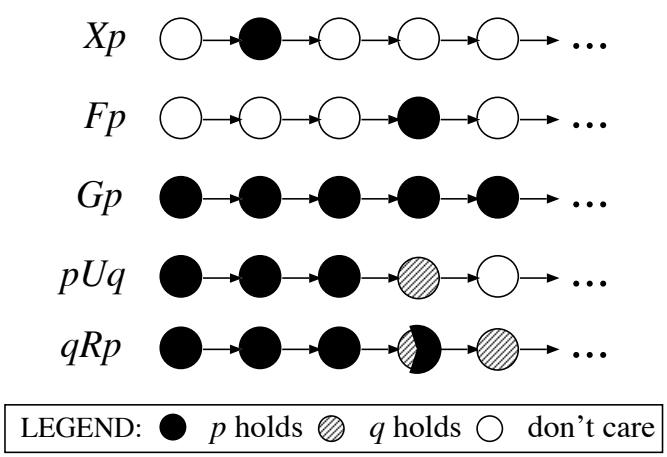

Figure 4.6: Linear Temporal Logic (from [Sim07])

logic synthesis [BSVMH84] and equivalence checking [GPB01]. Popular SMT solvers include Z3 [DMB08] and Yices [DM06].

- For **first order logic** we have *first order theorem provers* which aim to automate decidable fragments of first order logic (including propositional logic) whilst allowing user-guided proof for theorems expressed in the undecidable parts of the logic. A widely used first order theorem prover is ACL2 [KSM96]. SAT and SMT solvers can also provide automated reasoning for some fragments of first order logic.

- For **higher order logic** we have *higher order theorem provers* which aim to automate decidable fragments of higher order logic, whilst allowing user-guided proof for theorems expressed in the undecidable parts of the logic. Popular higher order theorem provers include PVS [ORS92], HOL [GM93] and Isabelle/HOL [NWP02].

- For **temporal logics** we have *model checkers* which provide fully automated reasoning for decidable theorems. Popular model checkers include SAL [dMOR+04], SPIN [Hol03] and NuSMV [CCGR99].

4.6 Automated Reasoning in the IC Design Flow

BSV supports *design by refinement*, in which electronic system level specifications are incrementally refined into implementations which can be synthesised to efficient

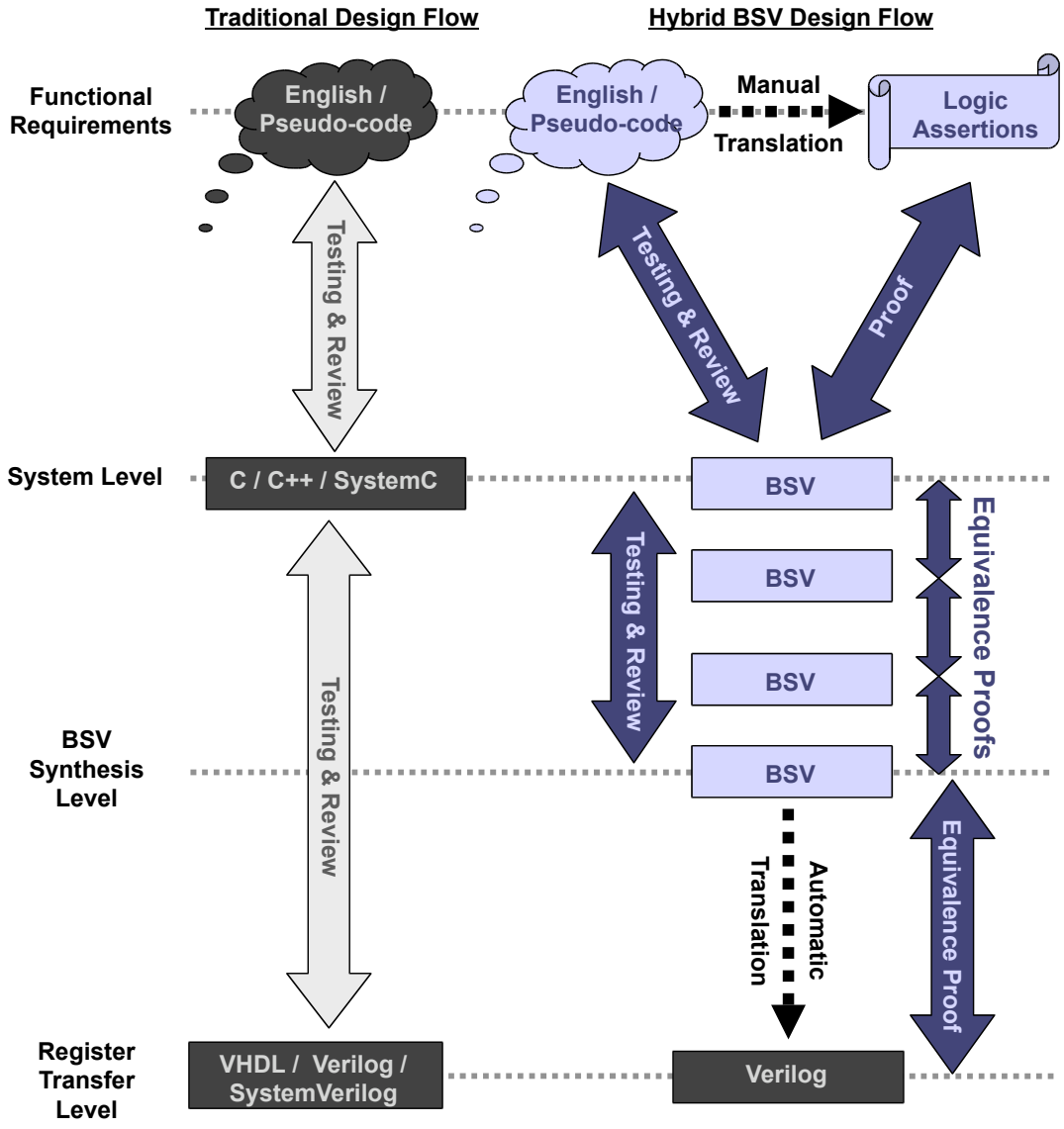

Figure 4.7: Automated Reasoning for BSV: A Conceptual Design Flow

hardware. Refinement-based design provides an excellent basis for the application of formal methods, as illustrated by the conceptual design flow in figure 4.7. This design flow satisfies a number of recommendations made by the International Technology Roadmap for Semiconductors for innovation over coming years [ITR09]:

1. **Formal verification early in the design flow** to identify errors in ESL specifications, when the cost of correction is lowest.
2. **Complete proof of functional equivalence** between ESL specifications and RTL designs.
3. **Design for verifiability** to render tractable goals (1) and (2):
   - ESL specifications should be structured in a hierarchical way, to enable the decomposition of complex systems into smaller blocks which are suitable for formal verification;
   - RTL designs should be derived from ESL specifications by incremental refinement, where each step in the refinement can be proven to preserve functional correctness.

ITRS sees these developments as integral to its suggested aggressive scale-up in formal verification, the timetable for which is shown in figure 4.8.

The design flow in figure 4.7 requires two kinds of proof:

1. **Functional verification** of ESL specifications against theorems expressed in logic.
2. **Proof of refinement** between increasingly detailed designs.

This dissertation focuses on the former: the latter is left as a topic for further work. We now review the current state-of-the-art in automated reasoning tools and techniques for high-level functional verification of hardware designs. This will lead us to a verification strategy for BSV, which will be partially implemented in the following chapters.

4.7 Functional Verification of ESL Specifications

Many languages express hardware as state machines; this is true for the BSV, and also for VHDL, Verilog and SystemC. Therefore, the task of verifying a given hardware

Year of Production	*2009*	*2010*	*2011*	*2012*	*2013*	*2014*	*2015*	*2016*
Design errors exposed using formal or semi-formal verification (%, versus simulation)	9.4	11.8	14.1	16.5	18.8	21.2	23.5	25.9
Portion of the design specification formalized for verifiability (%)	13.8	15.0	16.3	17.5	18.8	20.0	21.3	22.5
Year of Production	*2017*	*2018*	*2019*	*2020*	*2021*	*2022*	*2023*	*2024*
Design errors exposed using formal or semi-formal verification (%, versus simulation)	28.2	30.6	32.9	35.3	37.6	40	43	45
Portion of the design specification formalized for verifiability (%)	23.8	25.0	26.3	27.5	28.8	30.0	31.3	32.5

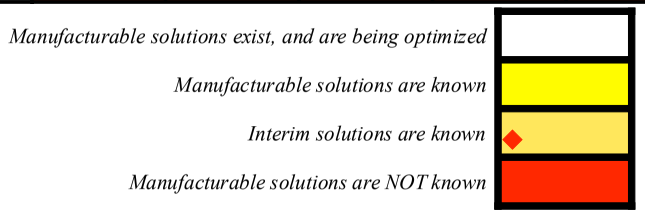

Figure 4.8: ITRS 2009 Formal Methods Roadmap. The first row sets targets for the increasing role of formal methods in functional verification, and the second row refers to the existence of a system level specification in a language which is suitable for formal proof; i.e. a semantically elegant language which is supported by the appropriate proof tool(s). Note that manufacturable solutions as of 2009 can only maintain the proposed schedule until 2012, and that there are *no known solutions* to maintain the schedule beyond 2016. All figures refer to designs which are newly developed in the relevant year, and exclude design components which are re-used from earlier projects. From [ITR09], table DESN6.

design in a given language often reduces to a more generic problem of verifying a certain type of state machine. Over recent years, automated reasoning has made great advances in its capacity to verify logic properties for complex hardware designs, fuelled by a stream of new techniques for verifying state machines. It is now used by a host of companies to verify substantial design components, as discussed in chapter 8. Interestingly, the software used by these companies is often built around off-the-shelf proof tools for verifying state machines.

4.7.1 How to Verify a State Machine

We can verify a state machine by specifying the desired properties in temporal logic and proving that they hold for all reachable states. When the state space is finite, we can determine whether the properties hold by exhaustive exploration of the reachable states. This is the idea of model checking. Model checking is attractive because it is fully automatic and returns counter-examples when theorems transpire to be false. These counter examples can then be used as a starting point to identify the underlying bugs that invalidate the theorems (as discussed in Rushby's illuminating paper [Rus00a]).

For systems with large state spaces, checking every state can become intractable. This is referred to as *state space explosion* and occurs in parameterised systems, systems with large numbers of concurrent interacting processes, and systems which contain data structures that can assume many different states (for example in *data paths*, which are circuits that transport data between storage locations). We must therefore look beyond model checking alone if we wish to verify systems with these attributes, all of which are common in hardware designs. Two popular methods are:

1. **State Space Reduction** – the state space is reduced to a size that can be model checked. A common technique is abstraction [CC77, CGP00, GS97].

2. **Compositional Reasoning** – a system's subcomponents are verified automatically (often using model checking with state space reduction) and the results are composed by user-guided deduction to verify larger systems [SJO+05].

There has also been a small amount of work on mathematical induction over state machines [BLO98], although it is challenging in practice because many state machine invariants are not *inductive*, and need to be *strengthened* by adding extra conditions.

4.7.2 Tools for Scalable State Machine Verification

We now investigate the possibilities for applying the above proof strategies for the verification of BSV designs. These strategies have been implemented in a number of interactive theorem provers, which combine powerful state-based verification techniques with the full spectrum of deductive reasoning (from complex mathematical induction[2] strategies down to simple operations such as skolemization and term rewriting) within proof environments which allow rapid experimentation and customisation. These theorem provers can be used for the verification of BVS designs if we can find a way to translate the designs of interest into a form which is accepted by such tools.

PVS

PVS is a higher order theorem prover which supports automatic deduction, model checking [RSS95], automatic predicate abstraction [SS99], automatic invariant strengthening [BLO98] (for mathematical induction over state machines) and SMT solving (via Yices [DM06]). All of these features are closely integrated into an interactive proof environment, and can be applied during interactive proof sessions by the invocation of proof strategies, or combined before a proof is attempted thanks to an expressive proof scripting language.

HOL

HOL is a higher order theorem prover, which is comparable in many ways to PVS. It also supports model checking and automatic abstraction [PTC+00, Amj03]. These capabilities have been applied to verify the Advanced Microcontroller Bus Architecture (AMBA) of ARM Limited [Amj06].

Forte and HOL-VOSS

Intel carries out formal verification with a proprietary tool called Forte [SJO+05], which is freely available for non-commercial use. Forte combines model checking with lightweight theorem proving in higher order logic. The model checker uses symbolic trajectory evaluation (STE) [Seg93] and supports a limited subset of Linear Temporal Logic, which has only the 'next state' and 'requires' path quantifiers. State space reduction is achieved by the use of a three valued logic, in which digital signals can be

[2] Mathematical induction – not to be confused with induction – is a form of deductive reasoning.

high, low or 'don't care'; this can dramatically reduce the number of cases that must be considered by the model checker.

Forte was heavily influenced by the earlier work of Joyce and Seger [JS93] on combining the HOL theorem prover with the VOSS STE model checker [Seg93]. In this system, VOSS establishes properties of manageable subcomponents of a design with model checking. These properties are then used in HOL, during interactive proof for the verification of larger design components.

Other Approaches

The ACL2 theorem prover supports deductive reasoning over first order logic, as well as fully automated SAT and Binary Decision Diagram (BDD) techniques. It has been used by Centaur and AMD to verify substantial hardware components [HR05, HS09, Obe99, Rus00b].

Dingel and Filkorn [DF95] apply the SVE model checker [FSS+94] together with the SEDUCT first order theorem prover [SN94] to verify LTL assertions for systems with transition relations specified in a VHDL-like imperative language. Their method uses data abstraction to verify large and unbounded systems.

Kurshan and Lamport [KL93] present an approach that combines the COSPAN model checker [HK90] with TLP [EGL93], a theorem prover based on the Temporal Logic of Actions. They verify a 64-bit multiplier by first model checking an 8-bit sub-component and extending the result to hold for an N-bit multiplier using TLP.

The Stanford Temporal Prover (STeP) [BBC+96, BBC+95] combines model checking with first order theorem proving to verify linear temporal logic specifications for finite-state and infinite-state transition systems.

Müller and Nipkow [MN95] combine model checking with the Isabelle theorem prover. Isabelle is used to form verified abstractions of finite and infinite state systems, which have sufficiently small state spaces to be verified by a prototype model checker.

4.8 Summary

We have discussed the practice of automated reasoning, from logics to proof tools, and reviewed the literature to find proof strategies of potential application to BSV. We have seen that a number of proof tools exist which combine powerful state-based verification techniques with the full spectrum of deductive reasoning, all within proof environments which allow rapid experimentation and customisation. Chapter 5 will

present a translation strategy for expressing BSV designs in the input logic of one such tool – the PVS theorem prover.

Chapter 5

Embedding Bluespec SystemVerilog in the PVS Logic

A subset of Bluespec SystemVerilog is embedded in the higher order logic of the PVS theorem prover. Owing to the clean semantics of BSV, application of monadic techniques leads to a surprisingly elegant embedding, in which hardware designs are translated into logic almost verbatim, preserving types and language constructs. The subset of BSV which is embedded includes module definition and instantiation, methods, implicit conditions, rule composition using methods from instantiated modules, and scheduling attributes.

5.1 Embedding BSV in Logic

Bluespec SystemVerilog is a formally inspired language, which evolved from research using Term Rewriting Systems (TRS) to produce hardware specifications which could be synthesised and formally verified [AS99]. In common with TRS, BSV is semantically elegant, which suggests that it is well suited for automated reasoning. To date, however, little work has been done to apply automated reasoning to BSV designs.

This chapter presents a shallow embedding of a subset of BSV in the specification logic of the PVS theorem prover. This provides access to the powerful automated reasoning capabilities of PVS (§4.7.2) including automatic deduction, model checking, automatic predicate abstraction, automatic invariant strengthening (for mathematical induction over state machines) and a versatile proof scripting language.

A novel application of monadic techniques allows BSV code to be translated *almost verbatim*, preserving types and language constructs (witness figures 5.2

and 5.3). The narrow semantic gap between BSV code and corresponding logic specifications reduces the risk of undetected errors in the BSV-to-PVS translation process. Furthermore, monadic embedding maintains the clean partitioning of state and functionality from BSV designs, which raises the possibility of applying compositional reasoning *at the source code level*, in which small design components are verified automatically (using model checking with abstraction, for example) and the results are composed by user-guided deduction to verify larger systems.

Monads have been used before to address the notoriously messy issue of verifying state-based computation with theorem proving. However, their application to BSV has yielded surprisingly clean results. BSV has its roots in a minimalist guarded action language (of the kind used to specify state machines for model checking) and was expanded into a fully-featured hardware language with strong influences from functional programming, which is essentially typed first-order or higher-order logic without quantification. As a result, we shall find that an environment such as PVS (which integrates model checking with higher order theorem proving) allows an almost verbatim translation of BSV source code to a monadic form which can be directly verified using standard proof strategies. These findings support the truism that a well designed language saves a great deal of elbow grease when automated reasoning is to be applied.

This chapter presents the first theorem prover embedding of BSV. In chapter 6, the embedding will be used to verify BSV designs using a combination of model checking and deductive reasoning. The application of automatic abstraction to BSV designs was unfortunately beyond the scope of this dissertation, although PVS was chosen above other proof tools because of its close integration of model checking and automatic abstraction [SS99]; capitalising on this would be a natural direction for further work.

5.2 Embedding the State of a BSV Module

The state type of a BSV module can be expressed as a PVS record. To take a simple example, the state type of a register (the module `mkReg` from §3.1) which holds elements of type `Bool` can be specified as:

```
BoolReg: TYPE = [#val: bool#]
```
PVS

The type of 'val' can be left unspecified by adding a type parameter to the containing PVS theory:

```
RegState[T: TYPE]: THEORY
 BEGIN

  Reg: TYPE = [#val: T#]

 END RegState
```
PVS

Other theories can import 'RegState' and generate Boolean registers, for example, with the shorthand 'Reg[bool]', which is expanded to 'RegState[bool].Reg' behind the scenes.

The states of more complex BSV modules can be defined in this way. For example, the state of a one element FIFO buffer (`mkFIFO1` from §3.2):

```
FIFO1State[T: TYPE]: THEORY
 BEGIN

  FIFO1: TYPE = [#notFull: bool, notEmpty: bool, val: T#]

 END FIFO1State
```
PVS

Note that a one-element FIFO does not actually need two boolean fields, because 'notEmpty' should always be '¬ notFull'. However, this is not the case for n-element FIFOs, so the two fields are defined for generality.

The states of modules which instantiate other modules can also be specified as records. For example, the module from the Peterson example of §3.2:

```
PetersonState: THEORY
 BEGIN

  IMPORTING RegState, FIFO1State

  PC: TYPE = {Sleeping, Trying, Critical}

  Peterson: TYPE =
          [#pcp : Reg[PC],
            pcq : Reg[PC],
            turn: Reg[bool],
            fifo : FIFO1[bool]#]

 END PetersonState
```

PVS

5.3 Embedding the Semantics of a BSV Module

The one-rule-at-a-time semantics for module instances (§3.1.1) can be expressed as a nondeterministic transition relation of the kind used to form Kripke structures for model checking (§4.4.1). First, rules must be specified as binary relations on the module state. For example:

```
myRule(pre, post: ModuleState): bool = ...
```

PVS

The internal structure of this function will be considered shortly; for now, note that it should evaluate to 'TRUE' whenever the 'post' state can be reached from the 'pre' state by a single application of the rule (i.e. whenever the guard evaluates to 'TRUE' for 'pre' and the action creates 'post' when applied to 'pre'). We can then express the one-rule-at-a-time semantics of a module instance as the disjunction of the module's rules:

```
transitions (pre, post: ModuleState): bool
  = rule_1 (pre, post) ∨ rule_2 (pre, post) ∨ ...
```

PVS

Recall from §3.1.1 that model checkers require transition relations to be either (i) left-total over the state type or (ii) left-total over the reachable states, depending on

the specific model checking algorithm used. Some model checkers provide *deadlock checkers* to verify this property for user-supplied transition relations: however, the PVS model checker does not. Furthermore, the PVS documentation does not elaborate on whether left-totality is required for all states or just the reachable states. Fortunately, it is relatively simple to ensure left-totality for all states by adding a *stutter term* to the transition relation, which maps every state to itself:

```
trans : VAR [[ModuleState, ModuleState] → bool]

left_total (trans) (pre, post: ModuleState): bool
  = trans (pre, post) ∨ pre = post
```

PVS

When applied to a transition relation, 'left_total' returns a new transition relation which contains a stutter term. Adding a stutter term gives the PVS specification a subtly different behaviour to the BSV code it represents, although this is OK for the proofs that we will conduct. Alternative solutions which do not have this effect are discussed in §6.2.

5.4 Embedding Rules: A Primitive Approach

Consider the following rule from the Peterson example of §3.2:

```
rule p_critical (pcp._read == Critical);
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule
```

BSV

The guard is a predicate on the state of a `mkPeterson` module instance, and the action changes the state of the module instance in some way. Recall from §3.2 that the FIFO method `enq` carries an implicit condition which will prevent `p_critical` from firing when `fifo` is full. This is enforced by allowing the guard of `p_critical` to *inherit* the implicit condition of `enq`. In general, we define a rule's *composite condition* to be the conjunction of its explicitly-declared guard and the implicit conditions of all the methods it invokes [Blu08]. A rule can only fire when its composite condition is satisfied.

With this in mind, we can represent `p_critical` with a PVS function of the form:

```
p_critical (pre, post: Peterson): bool =
    composite_condition (pre) ∧ post = action (pre)
```
PVS

Here, 'composite_condition (pre)' should evaluate to true if and only if the composite condition of `p_critical` evaluates to true in the BSV code, and 'action (pre)' should return a value of type 'Peterson' which is identical to 'pre', except with the updates that are prescribed by the action part of the BSV rule. Because module states are represented as records, 'composite_condition' will be a predicate on the fields of 'pre', and 'action' will make updates to the fields of 'pre'. For example, the above rule can be written as:

```
p_critical_primitive(pre, post: Peterson): bool =
    pre`pcp`val = Critical ∧ pre`fifo`notFull
    ∧ post = pre WITH [(fifo) := (#val := TRUE,
                                  notFull := FALSE,
                                  notEmpty := TRUE#),
                       (pcp) := (#val := Sleeping#),
                       (turn) := (#val := FALSE#)]
```
PVS

This is a straightforward way to express rules, and is suitable for model checking (as we shall see in chapters 6 and 7). We specify the state of a module with a record, and for each rule, we express the composite condition as a predicate over the fields of this record and the action as a record update. When we do this, we are effectively eliminating all of the methods that are used in the rule by expanding them in-place.

This approach seems quite simple, but it creates a problem. If we express a rule by fully expanding all of its method calls, we expose its full complexity. BSV provides the module and method constructs to avoid just this. If we specify a more complex module in this way (for example, one where rules and methods call methods which themselves call methods, all returning values and producing side-effects) we end up with a long-winded specification that bears little resemblance to the BSV code it represents. It then becomes difficult to provide assurance that the translation is accurate, which makes it difficult to rule out false positives when a property is proven, or conversely false negatives when a property is disproved.

Figure 5.1 shows the problem pictorially. We could automatically translate BSV code to an instance of the primitive PVS embedding (although no such tool exists at

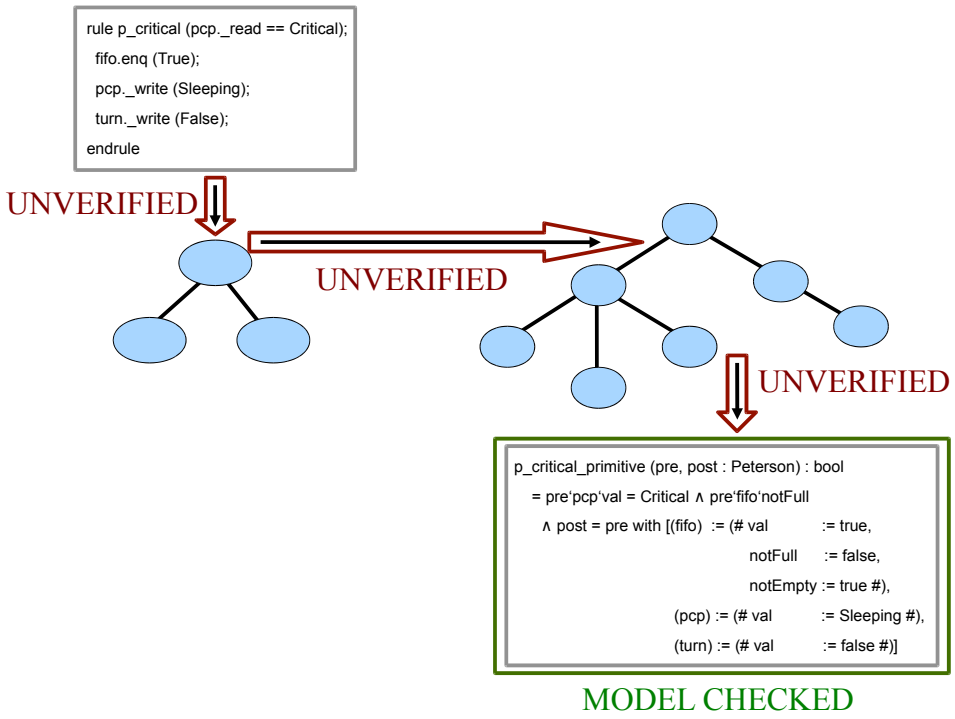

Figure 5.1: Verification of a Primitive Embedding in PVS

present) by parsing the BSV to get an abstract syntax tree (AST), and then transforming this AST to produce a new 'expanded' AST in which all method calls have been expanded in-place to expose their full complexity. We could then use this expanded AST to compile our primitive PVS embedding, and verify it with the proof strategies of PVS. However, the entire BSV-to-PVS translation process, which involves fully expanding *every* method call in-place, remains unverified. Of course, the same problem exists when compilation is carried out by hand, as it is currently.

We would like to have a PVS representation that relates back to the BSV system in a simple and transparent way. This can be achieved by applying the concept of monads to produce concise, high-level specifications.

5.5 Embedding Rules: A Monadic Approach

Monads [Mog91, Wad92a] provide a method of specifying 'impure' computations (such as state manipulation, nondeterminism and I/O operations) in functional languages. We will use them in the following sections to embed BSV rules and methods in the functional subset of the PVS logic.

```
wake_p: Rule = rule (pcp‘read = Sleeping)
                    (pcp‘write (Trying) ≫
                     turn‘write (FALSE))

wake_q: Rule = rule (pcq‘read = Sleeping)
                    (pcq‘write (Trying) ≫
                     turn‘write (TRUE))

grant_p: Rule = rule (pcp‘read = Trying
                      ∧ (turn‘read ∨ pcq‘read = Sleeping))
                    (pcp‘write (Critical))

grant_q: Rule = rule (pcq‘read = Trying
                      ∧ (¬ turn‘read ∨ pcp‘read = Sleeping))
                    (pcq‘write (Critical))

p_critical: Rule = rule (pcp‘read = Critical ∧ fifo‘enq_cond)
                        (fifo‘enq (TRUE) ≫
                         pcp‘write (Sleeping) ≫
                         turn‘write (FALSE))

q_critical: Rule = rule (pcq‘read = Critical ∧ fifo‘enq_cond)
                        (fifo‘enq (TRUE) ≫
                         pcq‘write (Sleeping) ≫
                         turn‘write (TRUE))

read_fifo: Rule = rule (fifo‘deq_cond)
                       (fifo‘deq)
```

PVS

Figure 5.2: Monadic Embeddings of the Peterson Rules in PVS

Before getting into the details, let us take a look at the results. Figures 5.2 and 5.3 show monadic embeddings of rules from the Peterson example and the Arbiter test bench. In contrast to the primitive embedding strategy of §5.4, the complexity of methods is factored out into monads. This yields rule specifications which are almost identical to the BSV rules they represent. The only non-trivial syntactic difference between BSV and the monadic PVS in these examples is the explicit inclusion of implicit conditions in the guards of rules (for example, note the expression 'fifo'enq_cond' in the 'p_critical' rule of figure 5.2 and 'arb'request1_cond' in the 'client1_req' rule of figure 5.3). With monadic PVS specifications being syntactically close to the BSV designs they represent, errors in the BSV-to-PVS translation process are discernible by inspection.

The monadic embedding strategy offers another potential advantage. Because a method is specified in a single place using a monadic function (rather than being expanded in-place whenever it is called, as occurs in the primitive embedding strategy) it may be possible to prove properties of methods, and use these properties to establish more complex properties of larger systems that use the methods, thus enabling compositional reasoning at the source code level. Compositional reasoning is not investigated in this dissertation, but presents an interesting possibility for further work.

```
client1_req: Rule = rule (arb'request1_cond) (arb'request1)

client2_req: Rule = rule (arb'request2_cond) (arb'request2)

client3_req: Rule = rule (arb'request3_cond) (arb'request3)

client1_hs: Rule = rule (arb'relinquish1_cond) (arb'relinquish1)

client2_hs: Rule = rule (arb'relinquish2_cond) (arb'relinquish2)

client3_hs: Rule = rule (arb'relinquish3_cond) (arb'relinquish3)
```

PVS

Figure 5.3: Monadic Embeddings of the mkTbArbiter Rules in PVS

5.5.1 Extensional Equivalence

Monadic rule specifications are actually functions; in the Peterson example, the type 'Rule' is a synonym for $\big[\big[\text{Peterson, Peterson}\big] \rightarrow \text{bool}\big]$. In fact, monadic rule specifications are extensionally equivalent[1] to their primitive counterparts from §5.4: *monads simply provide a more concise way of writing the same functions.* For example, if we fully expand all of the function calls in the definition of 'p_critical' in figure 5.2 (which can be done automatically in PVS with the `(expand*)` proof strategy [SORSC01]) we end up with the definition of 'p_critical_primitive' – a first order function which involves record updates and functions over record fields.

A BSV module can be specified in monadic form by: (i) defining a module state type and initial state; (ii) creating monadic specifications of its rules and methods; (iii) combining the monadic rule specifications using the 'left_total' function of §5.3 to form a transition relation for module instances. The PVS model checker fails when called directly on a monadic transition relation. (Interestingly, this is also true for the SAL model checker, as we shall see in chapter 7.) This is possibly because of the extensive use of higher order functions, which is uncommon in the formation of

[1] Functions are said to be extensionally equivalent if they provide the same outputs for all possible inputs.

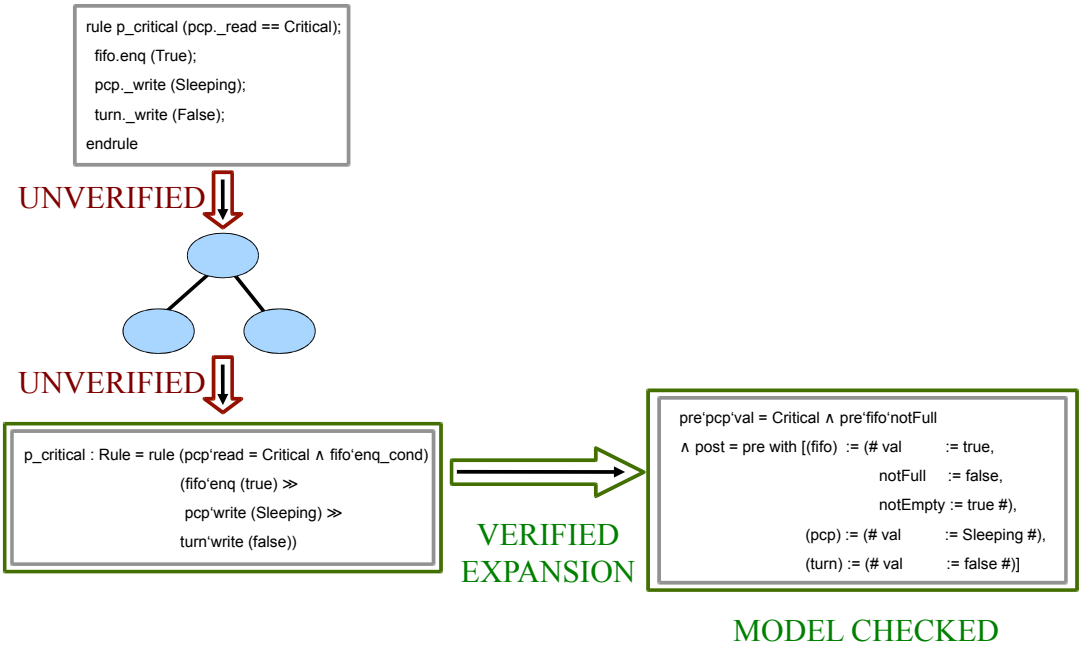

Figure 5.4: Verification of a Monadic Embedding in PVS

specifications for model checking. However, this problem can be circumvented in PVS with a small amount of deductive reasoning: the `(expand*)` proof strategy can automatically expand monadic transition relations into equivalent primitive transition relations, which are suitable for model checking. This verification strategy is shown in figure 5.4.

The remainder of this chapter elaborates on the technical details involved in producing monadic specifications of the kind shown in figures 5.2 and 5.3.

5.6 A Primer on Monads in PVS

This section implements the well-known *state monad* in the PVS logic. For further reading on monads, see [Bir98, OGS08]. The state monad is used to replicate side-effecting operations (which change the state of the environment in some way) with pure (side-effect free) functions. Imagine that we have a state of type '*S*', and we want to write a function that returns some value from the state, and also changes the state in some way. We can describe the type of this function with the following PVS theory:

```
StMonadType[S, A: TYPE]: THEORY
 BEGIN

  StMonad: TYPE = [S -> [A, S]]

 END StMonadType
```

PVS

A function of type 'StMonad' takes an instance of the state type, and returns a value of type 'A' and a new instance of the state; this is the well-known state monad from functional programming. When we want to specify a function that is side-effect free (i.e. it has no effect on the state) the output state will be identical to the input state. When we want to specify a function that is entirely side-effecting (so that no meaningful value is returned) the return value can have type 'Null':

```
NullType: THEORY
  BEGIN
    Null: TYPE+
    null: Null
  END NullType
```

PVS

This theory declares 'Null' to be a non-empty type ('TYPE+'), and declares an element of the type called 'null'.

The following theory defines a concrete state type, together with monadic functions which operate on it:

```
StMonadExample: THEORY
 BEGIN

  IMPORTING StMonadType, NullType

  MyRecord: TYPE = [#field1: int, field2: int#]

  read1: StMonad[MyRecord, int] =
      λ (myRec: MyRecord): (myRec`field1, myRec)

  read2: StMonad[MyRecord, int] =
      λ (myRec: MyRecord): (myRec`field2, myRec)

  write1(i: int): StMonad[MyRecord, Null] =
      λ (myRec: MyRecord): (null, myRec WITH [field1 := i])

  write2(i: int): StMonad[MyRecord, Null] =
      λ (myRec: MyRecord): (null, myRec WITH [field2 := i])

 END StMonadExample
```

PVS

'MyRecord' is a record type with two fields called 'field1' and 'field2'. Four monadic functions are defined for the 'MyRecord' type which return values from instances of 'MyRecord', and perform side-effects by 'writing over' the contents of the two fields:

- The function 'read1' reads the 'field1' field of a record. It has type:

  StMonad [MyRecord, int]

  which is a type synonym for:

  [MyRecord → [int, MyRecord]]

- The function 'read2' reads the 'field2' field of a record.

- The function 'write1 (i : int)' writes the value 'i' to the 'field1' field of a record.

- The function 'write2 (i : int)' writes the value 'i' to the 'field2' field of a record.

We would like to be able to compose these monadic functions *sequentially*, in the same way that we use semi-colons to compose statements in an imperative language. We can achieve this with *monad connectors*, which are defined in the following theory:

```
StMonadConnectors[S, A, B: TYPE]: THEORY
 BEGIN

  IMPORTING StMonadType

  ▷(m: StMonad[S, A], k: [A → StMonad[S, B]]): StMonad[S, B] =
      λ (s: S): LET (a, s1) = m(s) IN k(a)(s1);

  ▷▷(m: StMonad[S, A], n: StMonad[S, B]): StMonad[S, B] =
      m ▷ (λ (a: A): n)

 END StMonadConnectors
```

PVS

Both '▷' and '▷▷' are infix operators. Note the subtle semi-colon after the '▷' function; this is a delimiter, and is required for the PVS parser when defining infix operators [OSRSC01]. The functions '▷' and '▷▷' are actually quite intuitive to use, once you understand their internal workings.

The function '▷' is infix, taking two arguments:

1. A state monad '*m*' (for example 'read1' or 'read2' from 'StMonadExample' above) which takes an instance of the state type *S* and returns a pair of type $[A, S]$ (representing a value and a new state).

2. A function '*k*' from the type '*A*' to a state monad of type 'StMonad[S, B]' (for example 'write1' or 'write2' from above).

When these two arguments are applied to '▷', we have a new monad '$m \triangleright k$', which:

1. Takes an instance of the state type and passes it to '*m*', in order to retrieve a value and create a new state;

2. Applies the value obtained by '*m*' to '*k*', which returns a new state monad;

3. Passes the new state created by '*m*' to the new state monad created by '*k*'.

What is the purpose of this? It allows us to compose monadic functions sequentially:

```
MonadConnectorsExample: THEORY
 BEGIN

  IMPORTING StMonadExample, StMonadConnectors

  copy1to2: StMonad[MyRecord, Null] = read1 ▷ write2

 END MonadConnectorsExample
```
PVS

The function '▷▷' is another infix operator. It is similar to '▷', but takes two state monads, called '*m*' and '*n*'. The monadic function '*m* ▷▷ *n*' will take a value of the state type, apply it to '*m*', to get a value and a new state, throw away the value but keep the state and apply it to '*n*'. '▷▷' is similar to the semi-colon in imperative languages.

Monad connectors allow us to write 'imperative style' functions, such as the following:

```
MonadConnectorsExample2: THEORY
 BEGIN

  IMPORTING StMonadExample, StMonadConnectors

  swapValues: StMonad[MyRecord, Null] =
       read1      ▷ (λ (i: int):
       read2      ▷ (λ (j: int):
       write1(j)  ▷▷
       write2(i)))

 END MonadConnectorsExample2
```
PVS

The theory 'MonadConnectorsExample2' does not need to import the theories 'StMonadType' and 'NullType' because it imports 'StMonadExample', which imports them both. The function 'swapValues' has type:

StMonad [MyRecord, Null]

which is a type synonym for:

[MyRecord → [Null, MyRecord]]

It takes an instance of 'MyRecord' and returns a pair, in which the first element is 'null' and the second is a new record with the values of the fields swapped round.

5.7 A Monadic Representation of BSV Methods

We can represent BSV methods using a variation of the state monad. Consider the body of the rule `p_critical` from the `mkPeterson` module in §3.2:

```
fifo.enq (True);
pcp._write (Sleeping);
turn._write (False);
```
BSV

We have three method calls composed together in a single statement. Ignoring implicit conditions for the time being, the meaning we want to capture for the whole statement is that *an initial state is transformed independently by each of the three methods, and the changes made by each are combined to give a new state*. We can actually achieve the same effect by applying the methods sequentially, in the style that we created with the 'StMonad' and its connectors. We can apply the first method to get a partially updated state, then apply the second method to update this new state, and the third method to update the result. This is possible because the methods are conflict-free – the BSV compiler guarantees that no two method calls in a rule body will update the same element of state, so we need not worry about later method calls over-writing the updates made by earlier method calls. However, each method needs access to the state as it was immediately before the rule was executed, because earlier methods might update elements of state that later methods need to read. This suggests that we specify methods as instances of the type:

```
BSVMonadType[S, A: TYPE]: THEORY
 BEGIN

  BSVMonad: TYPE = [[S, S] → [A, S]]

 END BSVMonadType
```
PVS

Here, 'S' is the type of the module's state (in the case of `p_critical`, it is 'Peterson' from §5.2). 'A' is the type of some return value. Instances of 'BSVMonad' take two values of the state type (representing the state as it was immediately before the rule was executed, and a partially updated state) and return a value and a new instance of the state, with any additional updates added to those of the partially updated state.

5.7.1 Implicit Conditions

Recall from §5.4 that a method can contain an implicit condition, which is a predicate on the state of the containing module. Any rule which invokes a given method can only fire when the implicit condition of the method is satisfied (in addition, of course, to its own guard being satisfied). We say that the rule has a *composite condition*, which is the conjunction of its explicitly-declared guard and the implicit conditions of all the methods it invokes. In the monadic embedding strategy presented here, a method's implicit condition is embedded in PVS as a separate monad, and added to the guards of any rules that call the method. For example, see the implicit condition 'fifo'enq_cond' in the 'p_critical' rule of figure 5.2 and the implicit condition 'arb'request1_cond' in the 'client1_req' rule of figure 5.3.

5.7.2 Embedding the Methods of the mkReg Module

We can use this new monad type to define the methods of the `mkReg` module – namely, `_read` and `_write`. These methods do not have implicit conditions.

```
Reg[T: TYPE]: THEORY
 BEGIN

  IMPORTING BSVMonadType, NullType

  Reg: TYPE = [#val: T#]

  mkReg(init: T): Reg = (#val := init#)

  read: BSVMonad[Reg, T] = λ (pre, post : Reg): (pre'val, post)

  write(d: T): BSVMonad[Reg, Null] =
      λ (pre, post: Reg): (null, (#val := d#))

 END Reg
```

PVS

We use the name 'read' rather than '_read' because the latter is illegal in PVS. Notice that 'read' reads the contents of 'pre', but passes 'post' on. We have also defined a function 'mkReg', which constructs instances of the type 'Reg'.

5.7.3 Embedding the Methods of the mkFIFO1 Module

We can also embed the methods of the `mkFIFO1` module using the BSV monad. `mkFIFO` implements the `FIFO` interface:

```
interface FIFO #(type element_type);
  method Action enq(element_type x1);
  method Action deq();
  method element_type first();
  method Action clear();
endinterface: FIFO
```
BSV

The return type `Action` is analogous to the `Void` type in Java: it signifies that no meaningful value will be returned, so the method is entirely side-effecting. The method `first` has return type `element_type`; it is not denoted as an action, meaning that it is entirely side-effect free. Methods which perform a side-effect and also return a value are given the return type `ActionValue#(element_type)` [Blu08].

FIFO methods have the following behaviours:

- `enq` – places a value into the FIFO. This method has an implicit condition which prevents it from being called on a full FIFO.
- `deq` – removes the first value from the FIFO. It has an implicit condition which prevents it from being called called on an empty FIFO.
- `first` – returns the first value from the FIFO, without removing it. It has an implicit condition which prevents it from being called called on an empty FIFO.
- `clear` – clears all entries from the FIFO. It has no implicit condition.

Because `FIFO1` is a one-element FIFO, `clear` and `deq` have the same effect, apart from the fact that `deq` carries an implicit condition. However, for an n-element FIFO, `deq` removes a single element and `clear` clears the whole FIFO.

Figure 5.5 shows a monadic PVS embedding for the methods of the `mkFIFO1` module. Notice that the PVS function 'mkFIFO1' defines a valid initial state as any instance of the 'FIFO1' type for which the 'notFull' field is 'TRUE' and the 'notEmpty' field is 'FALSE'; the 'val' field is left undefined. The BSV methods `enq`, `deq` and `clear` are all split into two components:

```
FIFO1[T: TYPE]: THEORY
 BEGIN

  IMPORTING FIFO1State, BSVMonadType, NullType

  pre, post, fifo: VAR FIFO1

  mkFIFO1(fifo): bool = fifo`notFull ∧ ¬ fifo`notEmpty

  enq_cond: BSVMonad[FIFO1, bool] =
      λ (pre, post): (pre`notFull, post)

  enq(t: T): BSVMonad[FIFO1, Null] =
      λ (pre, post):
        (null, (#notFull := FALSE, notEmpty := TRUE, val := t#))

  deq_cond: BSVMonad[FIFO1, bool] =
      λ (pre, post): (pre`notEmpty, post)

  deq: BSVMonad[FIFO1, Null] =
      λ (pre, post):
        (null, (# notFull := TRUE, notEmpty := FALSE, val := pre`val #))

  first_cond: BSVMonad[FIFO1, bool] =
      λ (pre, post): (pre`notEmpty, post)

  first: BSVMonad[FIFO1, T] = λ (pre, post): (pre`val, post)

  clear: BSVMonad[FIFO1, Null] = deq

 END FIFO1
```

PVS

Figure 5.5: A Monadic Embedding of the mkFIFO1 Module

1. An implicit condition (e.g. 'enq_cond') which is a function of type 'BSVMonad [FIFO1, bool]'.

2. The method body (e.g. 'enq') which is a function of type 'BSVMonad[FIFO1,α]', where α is the return type of the method (where 'Null' in PVS corresponds to the `Action` return type in BSV).

5.8 Monad Connectors for the BSV Monad

We can compose instances of the 'BSVMonad' type sequentially, using the monad connectors '$\gg=$ ' (pronounced *bind*) and '$\gg$' (*seq*) which are slight variations of the functions '$\triangleright$' and '$\triangleright\triangleright$' from §5.6:

```
BSVConnectors[S, A, B: TYPE]: THEORY
 BEGIN

  IMPORTING BSVMonadType

  >>=(m: BSVMonad[S, A], k: [A -> state[S, B]]): BSVMonad[S, B] =
      λ (pre, post: S):
        LET (a, post1) = m(pre, post) IN k(a)(pre, post1);

  >>(m: BSVMonad[S, A], n: BSVMonad[S, B]): BSVMonad[S, B] =
      m >>= (λ (a: A): n)

 END BSVConnectors
```

PVS

Notice the subtle semi-colon, which was discussed in §5.6. Readers who are familiar with monads may be interested to know that the three monad laws have been stated and proven (in PVS) for these monad connectors, as part of the online code distribution [RL11].

5.9 Monad Transformers

Both of the functions '$\gg=$ ' and '$\gg$' combine monads that have *the same state type* (the type parameter *S*). For our BSV embedding, this means that we can only combine methods that operate on *a single instance of a single module type*. For example, we can construct the following 'negate' function:

```
TrivialFIFOFunction: THEORY
  BEGIN

  IMPORTING FIFO1

  negate: BSVMonad[FIFO1[bool], Null] =
      first >>= (λ (b: bool):
      deq >>
      enq (¬b))

END TrivialFIFOFunction
```
PVS

This elegant function negates the value held in an instance of the type 'FIFO1[bool]' (ignoring implicit conditions for the moment). However, we could not use our monad connectors to construct a comparable function which copied a value from one FIFO to another, or from a FIFO to a register. With this in mind, how could we embed the following?

```
fifo.enq (True);
pcp._write (Sleeping);
turn._write (False);
```
BSV

This is the body of the `p_critical` rule from the Peterson example (which is contained within a BSV module called `mkPeterson`). It calls methods for one instance of the `FIFO1` module and two instances of the `Reg` module. In reality, all of these module instances belong to a single instance of the `mkPeterson` module. We have already seen a PVS datatype that represents the state of a `mkPeterson` instance: the 'Peterson' type of §5.2.

Ideally, we would like to apply the monads that we have already created for the 'Reg' and 'FIFO1' states to the fields of an instance of the 'Peterson' state. We would like to *transform* our 'Reg' and 'FIFO1' monads into 'Peterson' monads. In fact, this is a common requirement when programming with monads, and has a standard solution in the form of *monad transformers*.

We could imagine a rather intuitive way to write the monad transformers we need. For the 'pcp' register, for example, we could write a function that took an instance of 'Peterson', extracted the 'pcp' field (which has type 'Reg[PC]'), applied the 'write' monad from the 'Reg' theory to get a new state, and returned a new 'Peterson' record,

with the 'pcp' field set to this new state. We say that this function *lifts* the 'write' monad of 'Reg' to operate on the 'Peterson' type. We could write a function like this whenever we wanted to lift a monad from one state type to another. However, this would involve writing boilerplate (unnecessarily repetitive code).

We can avoid boilerplate by factoring out the common functionality of monad transformers into more general functions. This is done in the following theory, which defines a type for monad transformers, together with a function for creating them:

```
MonadTransformer[R, S, A: TYPE]: THEORY
 BEGIN

  IMPORTING BSVMonadType

  Transformer: TYPE = [BSVMonad[R, A] → BSVMonad[S, A]]

  transform(get_R: [S → R], update_R: [[S, R] → S]): Transformer =
      λ (m: BSVMonad[R, A]):
        λ (pre, post: S):
          LET (val, post1) = m(get_R(pre), get_R(post)) IN
            (val, update_R(post, post1))

 END MonadTransformer
```

PVS

A function that has type 'Transformer' takes a monad over state type *R* and lifts it to become a monad over state type *S*. The function 'transform' constructs instances of the 'Transformer' type from two arguments:

- 'get_R' – a function that takes an instance of *S* and retrieves the appropriate field from it. For the 'pcp' field of the 'Peterson' type, this is simply:

  get_pcp (p: Peterson): Reg[PC] = p`pcp

- 'update_R' – a function that takes an instance of *S* and an instance of *R*, and writes the instance of *R* to the appropriate field in the instance of *S*. For the 'pcp' field of the 'Peterson' type, this is:

  update_pcp (p: Peterson, r: Reg[PC]): Peterson = p with [(pcp) := r]

The newly created 'Transformer' function then takes a monad called *m*, which operates on type *R*, and produces a new monad which operates on type *S*. The new monad takes a pair of elements of type *S* and uses 'get_R' to extract the appropriate *R* fields from them. It then applies *m* to the newly aquired *R* fields to get a value of type *A* and a new state of type *R*. Finally, it returns the value and places the updated *R* state back into the appropriate *S* state using 'update_R'.

These transformers are created for every 'Reg' and 'FIFO' field in the 'Peterson' type, and called 'pcpT', 'pcqT', 'turnT' etc. Once this has been done, the monadic functions for 'Reg' and 'FIFO' can be lifted to operate on the individual fields of a 'Peterson' instance. For example, the action component of the `p_critical` rule can be represented as:

```
my_peterson_monad: BSVMonad[Peterson, Null] =
    fifoT[Null](enq(TRUE)) ≫
    pcpT[Null](write(Sleeping)) ≫
    turnT[Null](write(FALSE))
```
PVS

With a little more work, this can be cleaned up to look almost identical to the actual BSV code. A record can be created for each individual register and FIFO instance, to hold the associated transformers, allowing monadic functions such as the following:

```
my_neater_peterson_monad: BSVMonad[Peterson, Null] =
    fifo`enq(TRUE) ≫
    pcp`write(Sleeping) ≫
    turn`write(FALSE)
```
PVS

In this example, the lifted 'Reg' and 'FIFO' monads are contained in records, which have the names of the fields within a 'Peterson' instance upon which they operate. In the case of 'Reg' instances, for example, these records are created with the following theory:

```
RegFunRecord[S, T: TYPE]: THEORY
 BEGIN

  IMPORTING Reg

  RegFunctions: TYPE = [# read : BSVMonad[S, T],
                          write: [T → BSVMonad[S, Null]]#]

  getRegFunctions (t_trans    : Transformer[Reg[T], S, T],
                   null_trans: Transformer[Reg[T], S, Null]): RegFunctions
        = (# read := t_trans(read),
             write := λ (t: T): null_trans(write(t)) #)

 END RegFunRecord
```

PVS

This theory is parameterised by the types S and T. S represents the type of a module which instantiates a register (e.g. 'Peterson') and T represents the type of elements that will be held by the register in question (e.g. 'bool' or 'PC').

5.10 Composing Monads to form Rules

We can use lifted monads directly in the guard if we overload the standard boolean and equality operators with functions over monads. For example:

```
∧ (m, n: BSVMonad[S, bool]): Monad[S, bool]
    = λ (pre, post : S): LET b1 = (m (pre, post))`1,
                             b2 = (n (pre, post))`1
                         IN  (b1 ∧ b2, post)
```

PVS

Here, the $\wedge$ operator used in the line '$\textsc{in}\ (b_1 \wedge b_2, \text{post})$' is the actual Boolean operator: b_1 and b_2 are both Booleans, which have been produced by the monads m and n. Placing monads in the guard allows predicates to be constructed in a readable way, having a concrete syntax which is similar to the actual guards in BSV.

Finally, when we have monadic specifications of a rule's guard and body, we can form a 'rule' that is a predicate over pairs of states. This is achieved with the function:

```
rule (guard: BSVMonad[S,bool]) (action: BSVMonad[S,Null]): Rule =
  (guard (pre, pre))`1 ∧ post = (action (pre, pre))`2
```
PVS

5.11 Experimental Results

The BSV examples of §3.2 and §3.3 have been translated into PVS using the primitive and monadic embedding strategies presented in this chapter. The full embeddings are available online [RL11].

Figure 5.2 shows the monadic embeddings of the Peterson rules. Furthermore, figure 5.6 shows three monadic rules together with their primitive counterparts, as well as extracts from the primitive and monadic transition relations and a lemma which asserts their equivalence. This lemma will be proven in chapter 6. Because the entire Peterson example was expressed in a single BSV module (`mkPeterson`), only FIFO and register methods required lifting with monad transformers.

The BSV arbiter example has a more complicated structure, being composed of two modules:

1. `mkArbiter` – which specifies the actual arbiter control circuit.
2. `mkTbArbiter` – a test bench which instantiates `mkArbiter` and invokes its methods.

In the PVS embedding, the `mkArbiter` module was embedded as a PVS theory, and lifted with monad transformers to allow instantiation within a separate theory that embedded the `mkTbArbiter` module.

Extracts from the monadic embedding of `mkArbiter` are provided in Figure 5.7. Note the guards of rules 'ack1_with_tok' and 'ack2_with_tok'. The module `mkArbiter` contains a scheduling attribute which specifies that 'ack1_with_tok' should always have priority over 'ack2_with_tok' – this is captured in PVS by referring explicitly to the guard of the former within the guard of the latter.

The state of a `mkTbArbiter` module is represented with the following record:

```
TbArbiter: TYPE = [# arb: Arbiter #]

mkTbArbiter: TbArbiter = (# arb := mkArbiter #)
```
PVS

```
wake_p  : Rule = rule (pcp‘read = Sleeping)
                      (pcp‘write (Trying) ≫
                       turn‘write (FALSE))

grant_p : Rule = rule (pcp‘read = Trying
                       ∧ (turn‘read ∨ pcq‘read = Sleeping))
                      (pcp‘write (Critical))

p_critical: Rule = rule (pcp‘read = Critical ∧ fifo‘enq_cond)
                        (fifo‘enq (TRUE)      ≫
                         pcp‘write (Sleeping) ≫
                         turn‘write (FALSE))

...

wake_p_primitive (pre, post: Peterson) : bool
  = pre‘pcp‘val = Sleeping
    ∧ post = pre WITH [(pcp)  := (# val := Trying #),
                       (turn) := (# val := FALSE #)]

grant_p_primitive(pre, post: Peterson): bool
  = pre‘pcp‘val = Trying ∧ (pre‘turn‘val ∨ pre‘pcq‘val = Sleeping)
    ∧ post = pre WITH [(pcp) := (# val := Critical #)]

p_critical_primitive (pre, post: Peterson) : bool
  = pre‘pcp‘val = Critical ∧ pre‘fifo‘notFull
      ∧ post = pre WITH [(fifo)  := (# val      := TRUE,
                                     notFull    := FALSE,
                                     notEmpty := TRUE   #),
                         (pcp)  := (# val      := Sleeping #),
                         (turn) := (# val      := FALSE    #)]

...

trans (pre, post: Peterson) : bool = wake_p    (pre, post) ∨
                                     grant_p   (pre, post) ∨
                                     p_critical (pre, post) ∨ ...

primitive_trans (pre,post: Peterson) : bool =
       wake_p_primitive    (pre, post) ∨
       grant_p_primitive   (pre, post) ∨
       p_critical_primitive (pre, post) ∨ ...

transitions_lem: LEMMA trans = primitive_trans
```

PVS

Figure 5.6: Extracts from the Peterson PVS Embedding

```
move_token: BSVMonad[Arbiter, Null] =
    tok1`read >>= tok2`write >>
    tok2`read >>= tok3`write >>
    tok3`read >>= tok1`write

ack1_with_tok_guard : BSVMonad[Arbiter, bool]
    = tok1`read ∧ req1`read
      ∧ ¬ (ack1`read ∨ ack2`read ∨ ack3`read)

ack1_with_tok : Rule = rule (ack1_with_tok_guard)
                            (move_token >>
                             ack1`write(TRUE))

ack2_with_tok_guard: BSVMonad[Arbiter, bool]
    = tok2`read ∧ req2`read
      ∧ ¬ (ack1`read ∨ ack2`read ∨ ack3`read)

ack2_with_tok : Rule = rule (ack2_with_tok_guard
                             ∧ ¬ ack1_with_tok_guard)
                            (move_token >>
                             ack2`write(TRUE))

request1_cond : BSVMonad[Arbiter, bool]
    = ¬ (req1`read ∨ ack1`read)

request1 : BSVMonad[Arbiter, Null]
    = req1`write(TRUE)

request2_cond : BSVMonad[Arbiter, bool]
    = ¬ (req2`read ∨ ack2`read)

request2 : BSVMonad[Arbiter, Null]
    = req2`write(TRUE)
```

PVS

Figure 5.7: Monadic Rules and Methods from the PVS Embedding of mkArbiter

The 'Arbiter' type is similar in structure to 'Peterson' of §5.2, except that its fields represent the nine registers instantiated in the `mkArbiter` BSV module of §3.3.

Figure 5.3 presents monadic embeddings of the `mkTbArbiter` rules. Note that 'arb' in this figure refers to a record of 'Arbiter' monads which have been lifted to operate on the 'TbArbiter' type (in the same way, for example, that 'pcp' and 'pcq' in figure 5.2 refer to records of 'Reg' monads which have been lifted to operate on the 'Peterson' type). Because the module `mkArbiter` contains rules which spontaneously alter the state of a module instance, the PVS embedding of `mkTbArbiter` must include these rules in its own transition relation:

```
pre, post: VAR TbArbiter

transitions (pre, post): bool =
    Arbiter.transitions (pre`arb, post`arb)
      ∨ client1_req (pre, post)
      ∨ client1_hs  (pre, post)
      ∨ client2_req (pre, post)
      ∨ client2_hs  (pre, post)
      ∨ client3_req (pre, post)
      ∨ client3_hs  (pre, post)
```

PVS

5.12 Shallow, Deep and Reflective Embedding

There are two approaches for embedding languages in the specification logics of proof tools [BGG+92, GMO06]:

1. **Deep Embedding** – the syntax of a language can be represented as a datatype in the logic, so that programs are instances of this type. The semantics of the language can then be specified with a function which transforms values of the "program" type.

2. **Shallow Embedding** – because the functional subset of a logic will have its own semantics, functions can be written which replicate the behaviour of programs in the language of interest.

This thesis is concerned with shallow embedding, which is often the simpler choice because it avoids the need to define the language's semantics explicitly – this is achieved by "piggy backing" on the semantics of the logic. In the case of BSV, we

have used the functional subset of the PVS logic to construct expressions which mimic BSV programs.

Shallow embedding is often the most convenient choice, but this convenience can come at a price. Shallow embedding allows reasoning about programs written in a language, but not about the language itself. Deep embedding, on the other hand, allows reasoning about both programs and language. The advantages of both approaches can be gained by producing a shallow embedding in a *reflective* language [GMO06] (that is, a language in which programs can be executed or manipulated by other programs). Reflection is supported, for example, by the ACL2 theorem prover [KSM96] and Intel's $reFL^{ect}$ language [GMO06]. The capabilities of deep, shallow and reflective embeddings are summarised in the following table:

Verification of...	Language Properties	Program Properties	Program Transformation
Shallow	×	✓	✓
Deep	✓	✓	✓
Reflective	✓	✓	✓

In further work, it may be interesting to produce deep and reflective embeddings of BSV in proof tools, in order to investigate the value of automated reasoning at the language level, and also to compare the efficiency of automated proof strategies for BSV programs expressed with deep, shallow and reflective embedding.

5.13 Summary

A shallow embedding has been presented of a non-trivial subset of Bluespec SystemVerilog in the higher order logic of the PVS theorem prover. The embedding uses monads and monad transformers to recreate BSV code almost verbatim in the PVS logic. Chapter 6 will investigate the verification of PVS specifications produced with this embedding strategy.

Chapter 6

Verifying BSV Designs with the PVS Theorem Prover

Automated reasoning is applied to PVS specifications which are produced with the monadic embedding strategy of chapter 5. In particular, two approaches are presented for the verification of temporal logic theorems: (i) monadic specifications are expanded within the PVS proof environment into equivalent primitive specifications, which are then model checked; (ii) separate monadic and primitive specifications are written in the PVS logic, and proven to be extensionally equivalent, allowing the former to be verified by rewriting and model checking. Scripts are written in the PVS proof strategy language; these implement the above approaches, to provide fully-automatic verification of temporal logic theorems concerning monadic specifications.

6.1 Model Checking in PVS

The PVS model checker requires that state machines are specified with:

1. **A state type** defined inductively from boolean and scalar types, using tuples, records or arrays over subranges.

2. **An initial state** or a predicate specifying a set of initial states.

3. **A transition relation** which is defined as a binary relation on the state type. As discussed in §4.4.1 and §5.3, this relation must be left-total over the state type.

The monadic and primitive embeddings presented in chapter 5 adhere to this format, as long as all types used in the BSV source code conform to the above state

```
ctlops[state: TYPE]: THEORY
 BEGIN

  u: VAR state
  f, g: VAR pred[state]
  N: VAR [state, state → bool]

  EX(N, f)(u): bool =  ...
  EG(N, f): pred[state] = ...
  EU(N, f, g): pred[state] = ...
  EF(N, f): pred[state] = ...
  AX(N, f): pred[state] = ...
  AF(N, f): pred[state] = ...
  AG(N, f): pred[state] = ...
  AU(N, f, g): pred[state] = ...

 END ctlops
```

PVS

Figure 6.1: Computation Tree Logic in PVS

type restriction (in which case, the representations of module states described in §5.2 will also conform). The automatic boolean abstraction capability of PVS can be used to reduce other types (such as integers, for example) to a model checkable form. Furthermore, automatic abstraction is well integrated with the PVS model checker (both can be applied with a single invocation of the proof strategy `(abstract-and-mc)` [SORSC01]). However, the present work is concerned only with model checking, leaving abstraction as a topic for further work.

Theorems for the PVS model checker can be written in CTL (§4.4.2), fair CTL or μ-calculus [RSS95]. In this work, theorems will be written in CTL. Fig. 6.1 provides extracts from the theory that defines CTL in PVS. The variable 'N' is a binary relation on the state type, and is used represent the transition relation. The variables 'f' and 'g' are predicates on the state type ('pred[A]' is a synonym for '[A → bool]').

The functions 'AX', 'AG', 'AF', 'EX', 'EG' and 'EF' all take a transition relation and a predicate (which should hold in some or all of the reachable states, as required by the particular function) and return a predicate on the *initial* state[1] which can be evaluated by the PVS model checker. For example, the model checker will evaluate

[1] This is the case for 'AX', although it is written differently to the other functions.

'AX (N, f) (init)' to 'TRUE' if and only if 'f' holds for all states which are reachable from 'init' by a single application of the transition relation 'N'.

The function 'AU' takes a transition relation 'N' and two predicates, 'f' and 'g'. It returns a predicate on the initial state which the model checker will evaluate to true if and only if 'f' holds along all paths from the initial state until 'g' holds – this is a standard definition of the AU operator, as discussed in §4.4.2. The function 'EU' takes the same arguments as 'AU', but returns a predicate which evaluates to true if and only if 'f' holds along at least one path from the initial state until 'g' holds.

6.1.1 Limitations of the PVS Model Checker

The concept of theorem proving with integrated model checking is very powerful, as demonstrated by Intel for example, who use their Forte system (§4.7.2, §8.4.1) to verify substantial design components. However, it should be noted that the PVS model checker is currently a prototype tool which lacks features that are common in more mature model checkers, such as deadlock checking, counter-example generation (the 'crown jewel' of the model checking paradigm [Rus00a]) and alternative back ends for explicit model checking, bounded model checking and so-on. These limitations can impact on the utility of PVS as a verification tool. For example, we saw in §5.3 that the lack of a deadlock checker in PVS necessitates a kludge to ensure that transition relations are left-total.

The research presented in this thesis has not been hindered by the above limitations because it is primarily focussed on the challenge of translating BSV to logic in a form which is suitable for model checking, irrespective of the particular tool. However, further work on the application of automated reasoning to BSV may benefit from a re-evaluation of the PVS model checker (which, one hopes, will mature over time) and an investigation of alternative tools (which were surveyed in §4.7.2).

6.2 Temporal Theorems for BSV Module Instances

We now use the computation tree logic of PVS to construct theorems for the monadic embeddings discussed in §5.11. These theorems will later be verified using the PVS model checker, together with a small amount of automatic deduction. Notice that temporal logic theorems are only ever written for left-total transition relations.

6.2.1 Theorems for Peterson's Protocol

We would like to establish the following properties for the monadic embedding of Peterson's protocol:

- **Deadlock freedom** – the system will never enter a state from which it cannot progress.

- **Safety** – at most one 'process' will have access to the 'critical' region at any one time.

- **Progress** – a process which enters the 'trying' phase will always eventually be granted access to the 'critical' region.

As we shall see, theorems (i) and (ii) can be captured in the CTL of PVS and verified with the PVS model checker. However, proof of theorem (iii) is complicated by the fact that we add a stutter term to transition relations to ensure left-totality (discussed in §5.3 and §6.1.1). Adding a stutter term gives the PVS specification a subtly different behaviour to the BSV code it represents – systems which stutter will not necessarily 'progress' in the sense defined above.

There are other ways to ensure left-totality. For example, one might add a boolean 'reachable' variable to the state of a module, and construct a transition relation in which 'reachable' states can progress to other 'reachable' states by valid BSV transitions, and all states can also progress to a designated 'unreachable' state which can stutter. Progress properties could then be proven for the reachable states (perhaps using the 'fair CTL' of PVS [ORR+96]). However, because the focus of this thesis is to investigate the broader concept of automated reasoning strategies for BSV, it implements the simple 'stutter' approach in order to avoid refocussing on how to address the idiosyncrasies of one particular tool (in this case, the lack of a deadlock checker in PVS – §6.1.1). In making this decision, we sacrifice proof of progress (although progress is, in fact, proven for both Peterson and Arbiter examples in chapter 7, using the SAL model checker).

We establish deadlock freedom with two theorems, the first of which states that an instance of the Peterson embedding will never *stutter*:

```
pre, post : VAR Peterson

no_stutter : THEOREM trans (pre, post) ⇒pre ≠ post
```
PVS

This can be proven with a single invocation of the powerful `(grind)` strategy. With this established, we assert in CTL that the left-total transition relation will never reach a state from which it cannot reach another (different) state:

```
p, p1, p2: VAR Peterson

deadlock_freedom: THEOREM
  mkPeterson(p) ⇒
   AG(left_total(trans), λ(p1): EX(left_total(trans), λ(p2): p1 ≠ p2)(p1))
      (p)
```
PVS

The meaning of this theorem is quite simple: having declared a predicate for the valid initial states of the Peterson module ('mkPeterson') as well as a transition relation ('trans') which is made left-total (as discussed in §4.4.1 and §5.3), we assert that for each state which is reachable from a valid initial state, there exists at least one other state to which it is not equal, and which can be reached by a single application of the transition relation (remember that the function 'left_total' ensures that all states can reach themselves by a single application of the transition relation).

Together, 'no_stutter' and 'deadlock_freedom' tell us that an instance of the Peterson BSV module will never enter a state from which it will not assuredly progress to another – different – state. The safety property can be expressed with a single CTL theorem on the left-total transition relation:

```
p: VAR Peterson

safety: THEOREM mkPeterson(p) ⇒ AG(left_total(trans), safe)(p)
      WHERE safe(p) =
                ¬ (p‘pcp‘val = Critical ∧ p‘pcq‘val = Critical)
```
PVS

This theorem asserts that for all states which are reachable from a valid initial state, 'pcp' and 'pcq' will not both be 'Critical'.

6.2.2 Theorems for a Round-Robin Arbiter

For the monadic arbiter embedding, we would like to establish three properties:

- **Deadlock freedom** – the system will never enter a state from which it cannot progress.

- **Safety** – at most one input will have access to the output at any one time. (This could also be called mutual exclusion).

- **Progress** – an input which raises a request will always eventually be granted access to the output.

As with the Peterson embedding, we sacrifice proof of progress for a simpler assurance of left-totality. Deadlock freedom is again specified with two theorems, which this time are written for the 'TbArbiter' type:

```
tb, tb₁, pre, post: VAR TbArbiter

no_stutter: THEOREM trans (pre, post) ⇒ pre ≠ post

deadlock_freedom: THEOREM
  AG (left_total (trans), λ (tb): EX(left_total (trans), λ (tb₁): tb ≠ tb₁)(tb))
     (mkTbArbiter)
```

PVS

As an aside, notice that 'mkTbArbiter' is a *unique value* of type 'TbArbiter', whereas 'mkPeterson' is a predicate which defines a *set of values* of type 'Peterson'. These are both acceptable ways to define the initial state(s) for the PVS model checker, so the choice is purely a matter of convenience. In the present work, 'mkPeterson' is a predicate because it uses 'mkFIFO', which was expressed as a predicate in §5.7.3, in order to avoid over-specifying the initial conditions of a FIFO buffer.

The above 'no_stutter' theorem can be proven with a single invocation of the `(grind)`. The 'deadlock_freedom' theorem will be proven below with model checking. Safety is defined as follows:

```
tb: VAR TbArbiter

safety: THEOREM AG (left_total(trans), safe)(mkTbArbiter)
    WHERE safe (tb) = ¬ (   tb‘arb‘ack1‘val ∧ tb‘arb‘ack2‘val
                         ∨ tb‘arb‘ack2‘val ∧ tb‘arb‘ack3‘val
                         ∨ tb‘arb‘ack3‘val ∧ tb‘arb‘ack1‘val)
```

PVS

6.3 Model Checking BSV Embeddings

When BSV modules are specified using the primitive embedding strategy of §5.4, temporal logic theorems can be proven with model checking (provided the modules have a manageable state space). However, as discussed in §5.4, instances of the primitive embedding strategy can be long-winded and bear little resemblance to the BSV code they represent, which makes it difficult to rule out errors in the BSV-to-PVS translation process.

Although monadic transition relations are written in the subset of the PVS logic that is accepted by the PVS model checker, the model checker fails to handle these specifications. This is possibly because of the extensive use of higher-order functions. Interestingly, the same problem occurs with the more mature SAL model checker, as we discuss in chapter 7.

Model checking *can* still be used to prove theorems concerning monadic transition relations, but a small amount of deductive reasoning must be applied first. As was mentioned in §5.5, the monadic embedding of a BSV module is extensionally equivalent to the primitive embedding of the same module. If we expand all of the function calls in a monadic embedding, we get a primitive embedding. Because of this, we can use either of two approaches to verify temporal logic theorems that refer to monadic transition relations:

1. **Proof with expansion** – we can expand all of the functions in a monadic embedding to get an equivalent primitive embedding. This can be done automatically inside the PVS proof environment with the proof strategy `(expand*)` [SORSC01]. We can then call the model checker on the resulting primitive embedding.

2. **Proof with rewriting** – we can construct a PVS theory which contains both a monadic embedding *and* an equivalent primitive embedding. We can then state

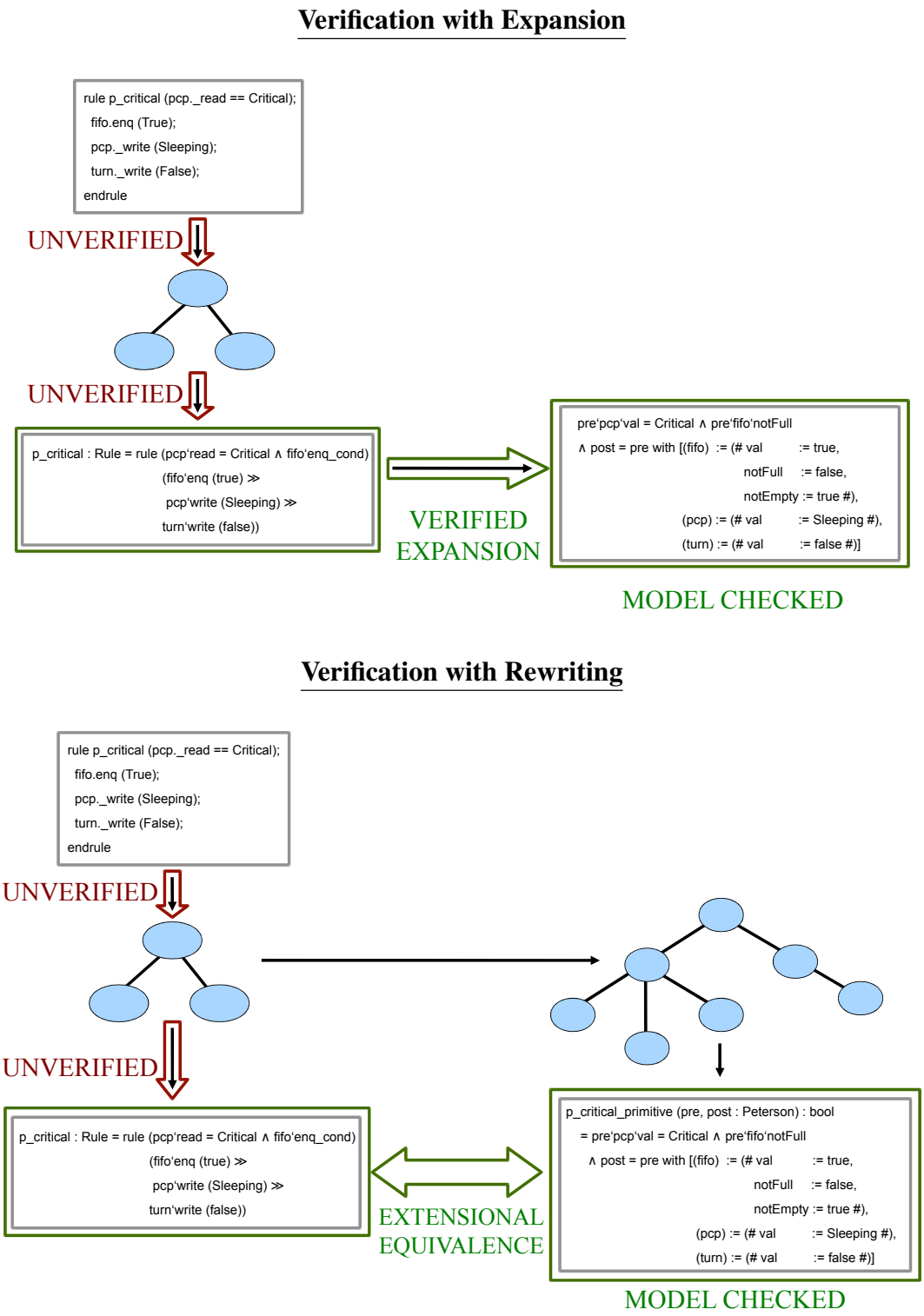

Figure 6.2: Verification Strategies for Monadic Specifications

that they are extensionally equivalent with a lemma:

```
transitions_lem: LEMMA trans = primitive_trans
```
PVS

The keyword 'LEMMA' is synonymous with 'THEOREM'. This lemma can be proven automatically with the deductive proof strategy `(grind-with-ext)` [Owr08]. Once this has been done, any occurrences of the monadic transition relation in a given theorem can be eliminated with the proof strategy `(rewrite "transitions_lem")`. This produces a new theorem concerning the primitive transition relation, to which model checking can be applied.

These two approaches are shown pictorially in figure 6.2. The blue trees represent abstract syntax trees (ASTs) which would be constructed during BSV-to-PVS translation (although the PVS specifications in this dissertation have been compiled by hand). In order to produce a monadic PVS embedding, a high-level AST would be extracted from the BSV and compiled straight to PVS, using monads to represent method calls, module instantiation and so-on. In order to compile from BSV to primitive PVS, the high-level AST would first need to be transformed, to eliminate the nodes which represent method calls, module instantiation and so-on, by expanding them in-place.

6.3.1 A Worked Example of Proof with Expansion

We now walk through a session in the PVS proof environment, in which deadlock freedom is proven for the monadic Peterson embedding, using the expansion approach introduced above. PVS employs the Emacs text editor as its user interface. A proof is initiated by placing the cursor on a theorem of interest and typing '`M-x pr`', whereupon a separate Emacs tab opens containing an interactive proof session. For the 'deadlock_freedom' theorem, we are presented with the following:

```
deadlock_freedom :
  |-------
{1}  FORALL (p: Peterson): mkPeterson(p) =>
        AG(left_total(trans), LAMBDA (p1):
            EX(left_total(trans),  LAMBDA (p2): p1 /= p2) (p1)
          ) (p)
Rule?
```
PVS Proof Environment

We proceed by expanding all of the functions in the monadic transition relation, in order to produce an equivalent primitive transition relation. This is achieved by repeatedly applying the `(expand*)` proof strategy [SORSC01], which expands the functions identified in its argument list:

```
Rule? (repeat (expand* "left_total" "trans" "wake_p" "wake_q"
                  < ... other monadic functions ... >))

this simplifies to:

  |-------
1    < deadlock_freedom theorem with a fully expanded
       transition relation >

Rule?
```

PVS Proof Environment

The `(repeat)` strategy applies `(expand*)` repeatedly, until no further changes can be made to the proof goal under consideration. We now have a temporal logic theorem concerning a fully-expanded transition relation, to which the PVS model checker can be applied:

```
Rule? (model-check)

this simplifies to:
{-1}  mkPeterson(p!1)
  |-------
{1}   p!1`fifo`notEmpty
{2}   p!1`fifo`notFull

Rule?
```

PVS Proof Environment

The model checker creates a trivial subgoal, which can be read as:

$$\texttt{mkPeterson(p!1)} \Rightarrow \texttt{p!1`fifo`notEmpty} \vee \texttt{p!1`fifo`notFull}$$

Here, `p!1` represents any value of type 'Peterson'[2], so the subgoal is saying that in any valid initial state, the FIFO will be not full or not empty (or both). We can discharge this subgoal to complete the proof, using the proof strategy `(grind)`:

[2] See (skolem) in [SORSC01].

```
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```
PVS Proof Environment

6.3.2 A Worked Example of Proof with Rewriting

We now prove 'deadlock_freedom' with the 'rewriting' approach discussed above. First we must prove 'transitions_lem', which can be done with a single application of `(grind-with-ext)` [Owr08]:

```
transitions_lem :
  |-------
[1]   trans = primitive_trans

Rule? (grind-with-ext)
Trying repeated skolemization, instantiation, if-lifting,
and extensionality,
Q.E.D.
```
PVS Proof Environment

The strategy `(grind-with-ext)` is similar to `(grind)`, but also performs extensionality. Once 'transition_lem' has been proven, it can be used as a rewrite rule in other proofs. For example:

```
deadlock_freedom :
  |-------
{1} FORALL (p: Peterson): mkPeterson(p) =>
       AG(left_total(trans), LAMBDA (p1):
         EX(left_total(trans), LAMBDA(p2): p1/=p2)(p1)
         ) (p)
Rule?
```
PVS Proof Environment

```
Rule? (rewrite "transitions_lem")
this simplifies to:
deadlock_freedom :
  |-------
{1} FORALL (p: Peterson): mkPeterson(p) =>
       AG(left_total(primitive_trans), LAMBDA(p1):
         EX(left_total(primitive_trans), LAMBDA(p2): p1/=p2)(p1)
         ) (p)
Rule?
```

PVS Proof Environment

This theorem can be proven with `(model-check)` and `(grind)`, as in §6.3.1.

6.4 Proof Strategies

PVS provides a proof strategy language [SORSC01], which can be used to automate recurrent proof tactics, such as the 'expansion' and 'equivalence' approaches discussed in §6.3. Strategies are defined with the following lisp-like expression:

```
(defstep name
         (required-parameters
           &optional optional-parameters
           &rest parameter)
    strategy-expression
    documentation-string
    format-string)
```

Parameter definition has the standard lisp syntax, with required and optional parameters, as well as a catch-all '*&rest*' parameter. The '*strategy-expression*' is the actual strategy, and is formed by combining existing proof strategies, using the control strategies `(try)`, `(if)` and `(let)`, which will be discussed shortly. The '*documentation-string*' is displayed in response to a `(help)` request, and the '*format-string*' is displayed whenever the strategy is invoked. User-defined strategies are placed into a file called '`pvs-strategies`', which is located in either the home directory or the working directory. This file is imported by PVS at the beginning of each proof session.

```
(defstep peterson-exp ()
  (repeat
    (expand* "trans" "left_total" "wake_p" "wake_q" "p_critical"
    "q_critical" "grant_p" "grant_q" "read_fifo" "rule" "pcq"
    "pcp" "turn" "fifo"  "getRegFunctions" "getFIFO1Functions"
    "pcqT" "pcpT" "turnT" "fifoT"  "transform" "read" "write"
    "enq" "enq_cond" "deq" "deq_cond" ">>" ">>=" "update_pcp"
    "update_pcq" "get_turn" "update_turn" "get_pcp"  "get_pcq"
    "get_fifo" "update_fifo" "NOT" "AND" "OR" "="  "run" "exec"))
  "A strategy to expand the monadic transition relation of
   Peterson.pvs"
  "(peterson-exp)")

(defstep tbarbiter-exp ()
  (repeat
    (expand* "left_total" "trans" "client1_req" "client1_hs"
    "client2_req" "client2_hs" "client3_req" "client3_hs" "rule"
    "arb" "getArbiterFunctions" "arbT" "get_arb" "update_arb"
    "transitions" "ack1_with_tok" "ack2_with_tok" "ack3_with_tok"
    "ack1" "ack2" "ack3" "ack1_with_tok_guard"
    "ack2_with_tok_guard" "ack3_with_tok_guard"
    "move_tok" "ack1_guard" "ack2_guard" "client1_req"
    "client1_hs" "client2_req" "client2_hs" "client3_req"
    "client3_hs" "arbiter_hs_1" "arbiter_hs_2" "arbiter_hs_3"
    "request1_cond" "request1" "request2_cond" "request2"
    "request3_cond" "request3" "relinquish1_cond" "relinquish1"
    "relinquish2_cond" "relinquish2" "relinquish3_cond"
    "relinquish3" "rule" "transform" "tok1" "tok2" "tok3" "req1"
    "req2" "req3" "ack1" "ack2" "ack3" "getRegFunctions" "tok1T"
    "tok2T" "tok3T" "ack1T" "ack2T" "ack3T" "req1T" "req2T"
    "req3T" "transform" "update_ack1" "update_ack2" "update_ack3"
    "update_req1" "update_req2" "update_req3" "update_tok1"
    "update_tok2" "update_tok3" "get_ack1" "get_ack2" "get_ack3"
    "get_req1" "get_req2" "get_req3" "get_tok1" "get_tok2"
    "get_tok3" "read" "notFull" "notEmpty" "write" "enq" "deq"
    "AND" "OR" "NOT" "=" ">>" ">>=" "exec"  "run"))
  "A strategy to expand the monadic transition relation of
   TbArbiter.pvs"
  "(tbarbiter-exp)")
```

PVS Strategy Language

Figure 6.3: Proof strategies to expand monadic transition relations.

Figure 6.3 presents strategies for converting monadic transition relations into equivalent primitive transition relations. The arguments of `(expand*)` are the names of functions used in the relevant monadic embedding. As seen in §6.3.1, the effect of `(repeat (expand* ...))` is to expand these functions repeatedly until no further changes can be made. Although monadic embeddings make extensive use of higher-order functions, they use no recursion whatsoever. Because of this, the expansion strategies in figure 6.3 will always terminate.

These strategies are relatively simple, but they spare the user from the tedium of providing a list of monadic functions each time an expansion is required. Instead, a single invocation of `(peterson-exp)` in the 'Peterson' theory or `(tbarbiter-exp)` in the 'TbArbiter' theory will effect a full expansion. An automatic BSV-to-PVS compiler could construct these strategies alongside the actual PVS theories, and write them to a '`pvs-strategies`' file in the relevant working directory.

As an aside, it should be noted that the order of the function names will affect the computational efficiency of this proof strategy. `(expand*)` iterates through the supplied list of function names, expanding each one that appears in the proof goal to generate a new goal. Care should be taken to ensure that top level functions are placed first in the list, so that their expansion will expose sub-functions, which in turn can be expanded as the list of identifiers is iterated through. In the worst case, when functions are listed in reverse order to their order of appearance, the number of repetitions of `(expand*)` will scale with the number of arguments that are passed to it. In-fact, it should be possible to find an ordering of the `(expand*)` argument list for which an application of `(repeat)` is not necessary, because PVS imposes a total order on function invocation[3]. However, practical experience suggests that this order can be difficult to find, and the computational expense of `(repeat)` is acceptable within the scope of the present work.

We can use the the control strategies of PVS to compose simple strategies such as `(peterson-exp)` and `(tbarbiter-exp)` into more sophisticated proof tactics. There are three control strategies, which derive from familiar constructs in software languages: `(if)`, `(let)` and `(try)`. We will make use of the latter, which has the following syntax:

`(try` *`step1 step2 step3`*`)`

[3]A given function can only call other functions that are defined above it in the containing PVS theory, or are imported from other theories. Furthermore, the 'IMPORTING' chain of a theory must form a directed acyclic graph, meaning that no theory can import itself, either directly or indirectly.

When a `(try)` strategy is invoked, it will apply *step1* to the current proof goal. If *step1* succeeds (i.e. generates new sub-goals), then *step2* will be applied to all of the new sub-goals. If *step1* fails (i.e. generates no change in the proof goal) then *step3* will be applied.

The following proof strategy automates the 'expand and model check' tactic of §6.3, which allows verification of temporal logic formulas concerning monadic BSV embeddings:

```
(defstep bsv-prove-with-exp (exp)
  (try (exp)
       (try (model-check)
            (grind)
            (skip))
       (skip))
  "" "" )
```

PVS Strategy Language

The documentation and format strings have been omitted for simplicity. The required parameter, `exp`, is a strategy to expand the monadic transition relation under consideration – in the 'Peterson' theory, for example, this will be `(peterson-exp)`. The strategy `(bsv-prove-with-exp exp)` will first apply `(exp)` to expand the monadic transition relation. If this produces a new sub-goal (which we expect to be a temporal logic formula concerning a fully-expanded transition relation) then the PVS model checker will be applied to it, and any sub-goals resulting from this application will be dispatched with `(grind)`.

The following proof strategy automates the 'rewrite and model check' tactic of §6.3:

```
(defstep bsv-prove-with-lem (lem)
  (try (rewrite lem)
       (try (model-check)
            (grind)
            (skip))
       (skip))
  "" "" )
```

PVS Strategy Language

This strategy differs from `(bsv-prove-with-exp)` only in its method of producing a tractable transition relation for model checking. Rather than expanding the monadic transition relation, the proof strategy `(rewrite)` is used to replace it with a pre-compiled primitive transition relation. In both 'Peterson' and 'TbArbiter' theories, a single invocation of `(bsv-prove-with-lem "transitions_lem")` will automatically prove the safety and deadlock freedom theorems.

6.5 Experimental Results

The temporal logic theorems presented in §6.2 can be proven automatically using the the proof strategies `(bsv-prove-with-exp)` and `(bsv-prove-with-lem)` of §6.4. For the Peterson example, we obtain the following results:

Theorem	Proof with Expansion	Proof with Rewrite	Deduction
deadlock_freedom	1.27 secs	0.25 secs	–
safety	0.85 secs	0.18 secs	–
transitions_lem	–	–	26.70 secs
no_stutter	–	–	0.24 secs

Here, 'proof with expansion' refers to an invocation of:

```
(bsv-prove-with-exp peterson-exp)
```

and 'proof with rewrite' refers to an invocation of:

```
(bsv-prove-with-lem "transitions_lem")
```

For the arbiter example, we have:

Theorem	Proof with Expansion	Proof with Rewrite	Deduction
deadlock_freedom	13.80 secs	0.42 secs	–
safety	7.01 secs	0.32 secs	–
transitions_lem	–	–	82.66 secs
no_stutter	–	–	0.78 secs

Here, 'proof with expansion' refers to an invocation of:

```
(bsv-prove-with-exp tbarbiter-exp)
```

and ‘proof with rewrite’ refers to an invocation of:

```
(bsv-prove-with-lem "transitions_lem")
```

These figures are indicative only, and are intended to give a feel for the relative computational requirements of the various proof techniques. Figures were obtained on a MacBook Pro with an Intel Core 2 Duo 2.53 GHz processor and 2 GB 1067 MHz DDR3 memory. The extra CPU time required to compile the primitive transition relation in the ‘rewrite’ approach could not be quantified because compilation is currently performed by hand. As discussed in §6.3, the extensional equality theorems (‘transitions_lem’) were proven in using the `(grind-with-ext)` strategy, and the ‘no_stutter’ theorems were proven with `(grind)`.

6.5.1 Proof with Expansion versus Proof with Rewriting

Proof with expansion offers the advantage that it is not necessary to compile a primitive embedding to PVS. On the other hand, an equivalence proof need only be carried out once for a given pair of monadic and primitive transition relations, and can be applied thereafter as a rewrite rule in all subsequent proofs, thus avoiding the need to expand the monadic transition relation.

Proofs of equivalence between primitive and monadic transition relations can easily be divided into smaller sub-proofs which are less computationally demanding. For example, we can write a theorem to express the extensional equivalence between a monadic rule and its primitive equivalent:

p_critical_lem : LEMMA p_critical = p_critical_primitive

PVS

(As mentioned in §6.3, the keyword ‘LEMMA’ is synonymous with ‘THEOREM’.) This can be proven using `(grind-with-ext)` and takes 0.28 seconds of CPU time. Once proven, this theorem can be applied as a rewrite rule during later proofs.

6.6 Summary

This chapter has presented two approaches for verifying monadic BSV embeddings with the PVS theorem prover:

1. *Proof with expansion*, in which monadic specifications are expanded within the PVS proof environment into equivalent primitive specifications that can then be model checked.

2. *Proof with rewriting*, in which separate monadic and primitive specifications can be compiled to logic and proven to be extensionally equivalent, allowing the former to be rewritten as the latter before model checking.

These approaches have been automated with strategies written in the PVS strategy language.

Chapter 7

Verifying BSV Designs with the Symbolic Analysis Laboratory

This short chapter evaluates the primitive and monadic embedding strategies of chapter 5 for compatibility with the SAL language and model checking suite. It is found that primitive embeddings can be expressed in the SAL language and can be model checked by the various SAL back ends. In contrast, monadic embeddings can be constructed in the SAL language but cannot be model checked. Deadlock freedom, safety and progress properties are verified for primitive SAL embeddings of the running BSV examples.

7.1 The Symbolic Analysis Laboratory

The Symbolic Analysis Laboratory (SAL) [dMOR+04] is a suite of model checking tools, comprising an expressive specification language and a number of independent back ends which include a well-formedness checker, a simulator, a deadlock checker, an automatic test generator and model checkers that implement symbolic, explicit, witness, bounded and infinite bounded model checking algorithms.

Chapters 5 and 6 have shown that BSV designs can be translated to the input logic of the PVS theorem prover, which provides access to the PVS model checker, in addition to a host of complementary proof techniques, including automatic abstraction. However, the PVS model checker is currently a relatively primitive tool, and there are a number of potential advantages in the ability to analyse BSV designs with a mature model checker such as SAL:

1. SAL provides counter-examples, simulation, automated test generation and a

deadlock checker, whereas the PVS model checker currently provides none of these.

2. The PVS model checker is symbolic, but it is often useful to have access to explicit, bounded, infinite bounded and/or witness model checking – SAL provides all of these, in addition to symbolic model checking.

7.2 Model Checking with SAL

The SAL language allows definition of nondeterministic state machines, using guarded action transition relations. In this way, its underlying model of computation is similar to BSV. A state machine is specified in a *module*, which declares a state type, a predicate to define the set of valid initial states and a transition relation. The following is a simple module, which is slightly adapted from [dM04]:

```
short: CONTEXT =
BEGIN
  Status: type = {ready, busy};
  main: MODULE =
  BEGIN
    INPUT request : bool
    OUTPUT status : Status
    INITIALIZATION
      status = ready
    TRANSITION
    [
      t1: status = ready AND request --> status' = busy
    []
      t2: not (status = ready) or not request --> status' = ready
    []
      t3: not (status = ready) or not request --> status' = busy
    ]
  END;
END
```

SAL

This example defines a *context*, which contains an enumerated type called `Status` and a module called `main`. The module `main` has an input called `request`, which is a boolean value that can be set by other modules, and an output called `status`, which is a value that can be set by the module `main`.

The transition relation declares three guarded actions: `t1`, `t2` and `t3`. The module evolves from a given state by choosing one guarded action for which the guard is true, and executing the associated action. As with the one-rule-at-a-time semantics of BSV, when more than one guard is true for a given state, a nondeterministic choice is made, with *no fairness condition* being placed on the choice. In the initial state, `status` is set to `ready`. Notice that `t2` and `t3` have the same guard, so that we have a nondeterministic choice whenever the predicate '`not (status = ready) or not request`' is true. Because SAL has no inbuilt concept of fairness, it is possible for `t3` to fire continuously once `status` is set to `busy`.

SAL allows properties to be specified in Linear Temporal Logic (§4.4.3). For example:

```
th1: theorem main |- G(request => F(status = busy));
```
SAL

We can see informally that this is true: whenever '`status = ready AND request`' evaluates to `true`, `t1` will fire (setting `status` to `busy`), and whenever the `status` is not `ready`, it must be `busy`. Hence, whenever `request = true`, the `status` will always be `busy` at some point in the future. We can prove `th1` with model checking, but we must first invoke the deadlock checker to prove that the transition relation is left-total, as discussed in §4.4.1:

```
% sal-deadlock-checker short main
ok (module does NOT contain deadlock states).
```
Command Line

SAL only requires transition relations to be total for the reachable states, rather than the entire state type. Having established left-totality, we can now prove `th1` with the SAL symbolic model checker:

```
% sal-smc short th1
proved.
WARNING: Your property is only true if it is deadlock free.
         You should run sal-deadlock-checker for that.
```
Command Line

7.3 Embedding BSV in the SAL Language

BSV modules can be embedded in the SAL language using both the primitive and monadic strategies presented in chapter 5. We can specify the states of modules using the approach presented in §5.2. In place of parameterised PVS theories, we use parameterised SAL contexts. For the Peterson example, we have:

```
Reg {T : type} : CONTEXT =
BEGIN
  State : type = [# val : T #];
END

FIFO1 {T : type} : CONTEXT =
BEGIN
  State : type = [# notFull  : bool,
                    notEmpty : bool,
                    val      : T   #];
END

PetersonState : CONTEXT =
BEGIN
  PC    : type = {Sleeping, Trying, Critical};
  State : type = [# pcp  : Reg{PC}!State,
                    pcq  : Reg{PC}!State,
                    turn : Reg{bool}!State,
                    fifo : FIFO1{bool}!State #];
END
```
SAL

```
[
   wake_p     : pcp.val = Sleeping
                --> pcp'  = (# val := Trying #);
                    turn' = (# val := false #)
[]
   wake_q     : pcq.val = Sleeping
                --> pcq'  = (# val := Trying #);
                    turn' = (# val := true #)
[]
   grant_p    : pcp.val = Trying
                and (turn.val or (pcq.val = Sleeping))
                --> pcp' = (# val := Critical #)
[]
   grant_q    : pcq.val = Trying
                and (not turn.val or (pcp.val = Sleeping))
                --> pcq' = (# val := Critical #)
[]
   p_critical : pcp.val = Critical and fifo.notFull
                --> fifo' = (# val     := true,
                               notFull  := false,
                               notEmpty := true     #);
                    pcp'  = (# val     := Sleeping #);
                    turn' = (# val     := false    #)
[]
   q_critical : pcq.val = Critical and fifo.notFull
                --> fifo' = (# val := true,
                               notFull := false,
                               notEmpty := true #);
                    pcq'  = (# val := Sleeping #);
                    turn' = (# val := true #)
[]
   read_fifo  : fifo.notEmpty
                --> fifo' = (# val := fifo.val,
                               notFull := true,
                               notEmpty := false #)
]
```

SAL

Figure 7.1: The Transition Relation of the Primitive Peterson Embedding

7.3.1 Primitive Embedding of Rules

BSV rules can be expressed as guarded actions in a SAL transition relation, using the primitive embedding strategy of §5.4, in which all method calls are expanded in-place. For the Peterson module of §3.2, this creates the transition relation shown in figure 7.1. As we shall see, these transition relations can be model checked with the various SAL back-ends.

7.3.2 Monadic Embedding of Rules

SAL has an expressive specification language which permits higher order functions, and hence supports the monadic embedding strategy of §5.5. For example, the following is a monadic SAL embedding of the `p_critical` rule from the Peterson example of §3.2 (also shown here for comparison):

```
p_critical = (pcp_read = Critical and fifo_enq_cond,
              bool_bool!seq (fifo.enq (true),
              PC_bool!seq   (pcp.write  (Sleeping),
                             turn.write (false))))
```

SAL

```
rule p_critical (pcp._read == Critical);
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule
```

BSV

This rule is expressed as a pair, in which the first element is a predicate on the state of the module and the second is a monad that can be applied to the state of the module to produce a new state. SAL does not support type inference or user-defined infix operators, so monadic specifications are a little less concise in SAL compared to PVS (for example, the standard monadic function `seq` is initialised once for each combination of types it is called with – in this case `bool_bool` and `PC_bool`) but the basic monadic form is preserved, including the use of monad transformers.

When monadic embedding is used, the transition relation is reduced to 'boilerplate' (simple, repetitive expressions), where the rules' monads are applied to the current

state:

```
[
   p_critical: p_critical.1 --> s' = bool_!exec (p_critical.2,s)
[]
   q_critical: q_critical.1 --> s' = bool_!exec (q_critical.2,s)
[]
    ...
]
```
SAL

The function `bool_!exec` takes a pair, where the first element is a monad (for example, `p_critical.2`) and the second element is the state of the module. It then applies the state to the monad and returns the updated state.

The various SAL back ends (deadlock checker, symbolic model checker etc.) will accept monadic specifications, but fail to reduce them to a tractable BDD form (typically terminating with a segmentation fault). Whilst extensive use of higher-order functional programming (for example, the implementation of monads and monad transformers) is not fundamentally incompatible with the algorithms used for model checking, it is presently uncommon in the formation of model checking transition relations, and consequently appears to be poorly supported by the various SAL back ends. Because the monadic embedding approach did not produce workable SAL specifications, a discussion is not provided of the various kludges required to support monadic programming in the SAL language. However, a complete monadic SAL embedding of the Peterson example is provided online for reference [RL11].

7.4 Verifying BSV-to-SAL Translation

We have seen that primitive embeddings of BSV modules can be expressed in the SAL language and processed by the various SAL back ends. This creates the possibility of a hybrid verification approach for BSV, which combines the specialist model checking capabilities of SAL with the versatility of a deductive tool such as PVS. Primitive SAL embeddings could be verified with model checking, and the deductive tool could establish extensional equivalence between primitive and monadic embeddings, and possibly also apply techniques such as automatic abstraction and compositional reasoning.

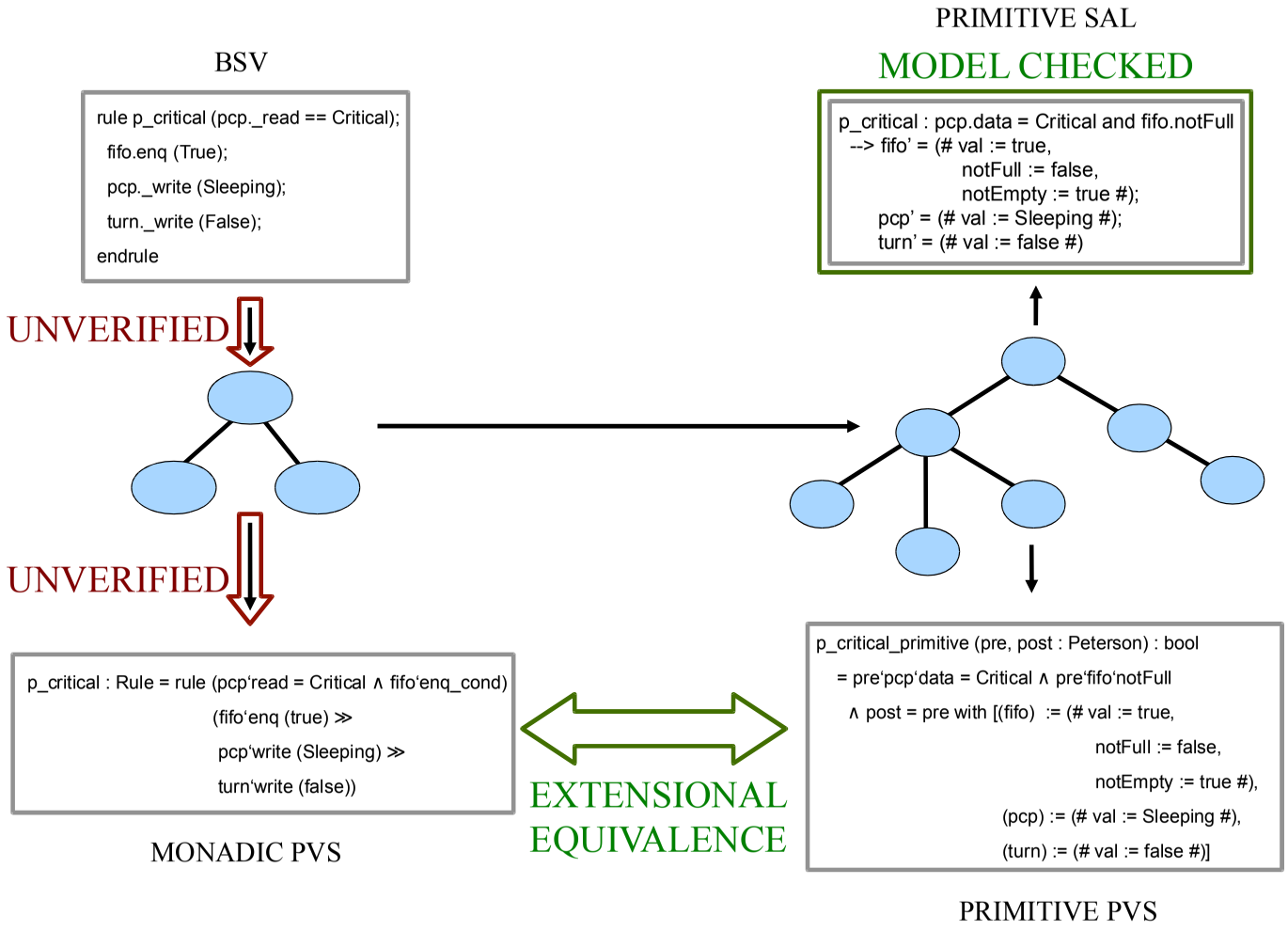

Figure 7.2: BSV Verification with PVS and SAL

This approach is outlined in figure 7.2. A BSV-to-PVS/SAL translation tool would parse BSV to produce an abstract syntax tree (call it the BSV AST) which could be converted to a monadic PVS embedding. The BSV AST would then be converted into a more complex AST by expanding out all method calls, module instantiations and so-on. This new AST (call it the expanded AST) could then be used to produce the nearly-identical primitive embeddings in PVS and SAL. The monadic and primitive PVS embeddings could then be proven extensionally equivalent, and the SAL embedding could be verified with model checking.

To give an idea of how the two PVS embeddings align with BSV and the SAL embedding, figure 7.3 shows the BSV rule `p_critical` along with its embeddings in PVS and SAL. Notice the strong syntactic resemblance between the BSV rule and its monadic PVS embedding, and also between the primitive PVS embedding and the SAL embedding.

7.5 Verifying Peterson's Protocol

SAL supports linear temporal logic, which can be used to construct safety and progress theorems for the primitive Peterson embedding:

```
safety: THEOREM
  System |- G (NOT(pcp.val = Critical AND pcq.val = Critical));

progress: THEOREM
  System |-  (G(pcp.val = Trying => F(pcp.val = Critical)))
         and (G(pcq.val = Trying => F(pcq.val = Critical)));
```

SAL

The safety theorem states that processes p and q will never simultaneously have access to the 'critical' section. The progress theorem states that a `Trying` process will always eventually gain access to the 'critical' section.

The SAL deadlock checker establishes deadlock freedom for the primitive Peterson embedding in 0.1 seconds. Once this has been done, the safety and progress theorems can be proven by the SAL symbolic model checker in 0.1 and 0.2 seconds respectively. Extensional equivalence between primitive and monadic PVS embeddings of the Peterson example was proven in chapter 6. As with the experimental results of §6.5, these figures are indicative only.

The BSV rule:

```
rule p_critical (pcp._read == Critical);
  fifo.enq (True);
  pcp._write (Sleeping);
  turn._write (False);
endrule
```
BSV

A monadic embedding in PVS:

```
p_critical: Rule = rule (pcp‘read = Critical ∧ fifo‘enq_cond)
                        (fifo‘enq (TRUE) ≫
                         pcp‘write (Sleeping) ≫
                         turn‘write (FALSE))
```
PVS

A primitive embedding in PVS:

```
p_critical_primitive(pre, post: Peterson): bool =
    pre‘pcp‘val = Critical ∧ pre‘fifo‘notFull
    ∧ post = pre WITH [(fifo) := (#val := TRUE,
                                  notFull := FALSE,
                                  notEmpty := TRUE#),
                       (pcp) := (#val := Sleeping#),
                       (turn) := (#val := FALSE#)]
```
PVS

A primitive embedding in SAL:

```
p_critical : pcp.val = Critical and fifo.notFull
             --> fifo' = (# val     := true,
                            notFull  := false,
                            notEmpty := true     #);
                 pcp'  = (# val     := Sleeping #);
                 turn' = (# val     := false    #)
```
SAL

Figure 7.3: A BSV Rule and its Embeddings in PVS and SAL

7.6 Verifying an Arbiter Control Circuit

Figure 7.4 presents extracts from the primitive SAL embedding of the arbiter test bench. Safety and progress are defined as follows:

```
safety: THEOREM System |- G(NOT(   (arb.ack1.val AND arb.ack2.val)
                                OR (arb.ack2.val AND arb.ack3.val)
                                OR (arb.ack1.val AND arb.ack3.val)));

progress: THEOREM System |- (G(arb.req1.val => F(arb.ack1.val)))
                         AND (G(arb.req2.val => F(arb.ack2.val)))
                         AND (G(arb.req3.val => F(arb.ack3.val)));
```
SAL

Deadlock checking requires 0.2 seconds, whilst proof of safety and progress require 0.2 and 0.3 seconds respectively.

We can also verify that at least one input will always eventually raise a request, from any given future state:

```
infinite_requests: THEOREM System |- G(F(   arb.req1.val
                                         OR arb.req2.val
                                         OR arb.req3.val));
```
SAL

This theorem is proven in 0.2 seconds.

7.7 Summary

BSV designs have been translated to the SAL language using the monadic and primitive embedding strategies of chapter 5. It has been found that primitive embedding yields model checkable SAL specifications, thus providing access to the extensive suite of verification tools provided by SAL. Furthermore, a technique has been presented for verifying 'BSV to primitive SAL' translation with the PVS theorem prover, by producing both monadic and primitive embeddings in the PVS logic, and proving them to be extensionally equivalent using the powerful deductive capabilities of PVS.

Whilst PVS provides an integrated model checker, there are a number of advantages in compiling to a mature model checker such as SAL:

```
TbArbiterPrimitive : context = BEGIN
  System : MODULE = BEGIN

    LOCAL arb : Arbiter!State,
          ack1_with_tok_guard : bool,
          ack2_with_tok_guard : bool,
          ack3_with_tok_guard : bool,
          ack1_guard : bool,
          ack2_guard : bool,
          ack3_guard : bool

    INITIALIZATION arb = Arbiter!mkArbiter;

    DEFINITION

    ack1_with_tok_guard
      = arb.tok1.val and arb.req1.val
        and not (arb.ack1.val or arb.ack2.val or arb.ack3.val);

    ack2_with_tok_guard
      = arb.tok2.val and arb.req2.val
        and not (arb.ack1.val or arb.ack2.val or arb.ack3.val);

   ...

    TRANSITION
    [  ack_1_with_tok : ack1_with_tok_guard
                        --> arb'.ack1 = (# val := true #);
                            arb'.tok1 = (# val := arb.tok3.val #);
                            arb'.tok2 = (# val := arb.tok1.val #);
                            arb'.tok3 = (# val := arb.tok2.val #)
    [] ack_2_with_tok : ack2_with_tok_guard
                        and not ack1_with_tok_guard
                        --> arb'.ack2 = (# val := true #);
                            arb'.tok1 = (# val := arb.tok3.val #);
                            arb'.tok2 = (# val := arb.tok1.val #);
                            arb'.tok3 = (# val := arb.tok2.val #)
    [] ...
    ]
  END;
END
```

SAL

Figure 7.4: Extracts from the Primitive Arbiter Embedding

1. SAL provides counter-examples, simulation, automated test generation and a deadlock checker, whereas the PVS model checker currently provides none of these.

2. The PVS model checker is symbolic, but it is often useful to have access to explicit, bounded, infinite bounded and/or witness model checking – SAL provides all of these, in addition to symbolic model checking.

Chapter 8

Related Work

A discussion is presented of work related to the BSV embeddings of earlier chapters, focussing in particular on the application of automated reasoning to: monadic functions, guarded action languages, functional and functionally-embedded hardware languages, and ad hoc *hardware languages. BSV is found to present unique challenges for the application of automated reasoning, being more complex than most guarded action languages, and exceptional amongst hardware languages in its choice of guarded actions as the underlying model of concurrency.*

8.1 Monads for Specification and Proof

Monads were introduced into functional programming by Moggi and Wadler [Mog91, Wad92a] as a method of specifying 'impure' computations, such as state manipulation, non-determinism and I/O operations. They were first used in association with theorem proving by Filliâtre [Fil03]. Subsequently, monads have been used a number of times in the context of theorem proving, most commonly to represent side-effects in the functional subsets of theorem prover logics for the abstract specification of state-based systems [BKH+08, FM10, HMW05, SB07], but also to produce shallow embeddings for subsets of the C language and Java [JP00, KM02].

The present work is unique in its application of monads for the embedding in logic of a hardware description language, and also in its support for the verification of a general class of temporal logic theorems over monadic specifications using combined model checking and mechanical deduction. The more common verification approach is to use Hoare logic and weakest precondition calculus, which are popular for the verification of software programs and were introduced to monadic verification

by Moggi [Mog91]. Exceptions to this trend are Bishop *et al.* [BFN+06], who employ a special-purpose path checker to establish that simulation runs comply with monadic HOL specifications, and also Sprenger and Basin [SB07], who verify monadic functions with a custom-embedding of linear temporal logic (LTL) in Isabelle/HOL, along with proof rules which reduce temporal reasoning to pre-condition/post-condition reasoning.

Of the existing literature concerning monads and theorem proving, research on embedding C and Java is most closely related to the present work, having been focused on the faithful representation of language semantics, in a form which is conducive to efficient automated reasoning. However, there are three important distinctions:

1. C and Java are *ac hoc* languages with complex semantics, which were developed without a view to formal verification: in contrast, BSV was designed with intentionally elegant semantics.

2. C and Java are software languages, meaning that an embedding must address complications such as non-termination, exceptions, memory management and so-on. BSV, being a hardware language, does not include such complexities.

3. C and Java have fundamentally different underlying models of computation when compared to BSV (imperative and object-oriented respectively, versus the guarded action model of BSV).

Krstic and Matthews [KM02] embed a subset of C in Isabelle/HOL for the purpose of verifying binary decision diagram (BDD) algorithms. Due to the semantic complexity of C, they omit underlying implementation details such as the concrete representation of state and garbage collection. Correctness theorems are specified in higher order logic and proven with interactive deduction. Jacobs and Poll [JP00] specify the denotational semantics of a subset of Java in PVS by defining a state monad with exceptions. This embedding is used by Huisman *et al.* [HJvdB01] to verify a safety property for the Vector class of the Java standard library, using Hoare Logic and interactive proof.

8.2 Guarded Action Languages

BSV is unique as a guarded action language which supports general purpose hardware design and synthesis. However, guarded actions are widely used in the field of formal

methods for specifying concurrent systems. They provide a natural way to express transition relations for model checking, and are also used as a basis for deductive proof in formalisms such as TLA$^+$ [Lam02], UNITY [CM88] and Event-B [AH07].

8.2.1 Bluespec SystemVerilog

Singh and Shukla [SS08] provide the only previous investigation into automated reasoning for BSV. They embed a subset of BSV in Promela, the specification language of the SPIN model checker. The subset of BSV is restricted to modules which instantiate only base modules such as registers. This essentially limits the syntax of BSV to the subset which is common with Promela. In contrast, the present work employs monadic techniques to specify advanced language features, and verifies the resulting logic expressions with a combination of model checking and automatic deduction.

Stoy *et al.* [SSA01] use hand-proof to verify an implementation of the cache-coherence protocol *Cachet*, which was constructed in TRS (the predecessor of BSV). In the same paper, they provide a separate specification of Cachet in the Temporal Logic of Actions [Lam94], which they translate into the PVS logic and verify with the PVS theorem prover.

8.2.2 TLA$^+$

TLA$^+$ is a logic for specifying concurrent systems and proving properties about them. Systems are typically described with:

1. A state type;
2. An initial state predicate;
3. A collection of state transitions, which are expressed as first order logic formulas on primed and unprimed variables;
4. Temporal formulas written in a subset of LTL which assert safety and liveness properties.

Transitions can be specified as guarded actions. For example, consider the following first order formula on integer variables x and y:

$$\underbrace{x+y \leq 100}_{guard} \;\wedge\; \underbrace{x' = x+y \;\wedge\; y' = y+1}_{action}$$

However, first order logic also permits more abstract transition formulas. For example:

$$x + y' \leq 100 \wedge \exists z \in 0..10 : y' = y + z$$

TLA$^+$ has a module structure which is similar to that of BSV. Modules contain state, transition formulas and temporal formulas, and can also instantiate other modules. The internal state of a module instance can be accessed and updated by invoking parameterised state formulas.

TLA$^+$ originated as a hand-based formalism, but several mechanical proof tools have been developed to support model checking and deductive reasoning for various subsets of the logic.

The TLA$^+$ Proof System

The TLA$^+$ Proof System (TLAPS) [CDLM08, CDLM10] checks the validity of manually-constructed deduction proofs. This is achieved by decomposing the proofs into collections of proof obligations, which are sent to the Zenon first-order tableau prover [BDD07] for automatic verification and, if this is unsuccessful, to Isabelle for interactive verification. TLAPS can be used to verify non-temporal theorems as well as the temporal theorems which express safety, but not liveness.

Module instances are expanded in-place, in a manner that is comparable to the 'primitive' embedding strategy for BSV, which was presented in chapter 5. The disadvantages of in-place expansion were discussed in chapter 5, and addressed for BSV by monadic embedding. However, it is unclear whether monadic techniques would be applicable to TLA$^+$, owing to its expression of transitions as first order logic formulas rather than the more restricted guarded actions of BSV. It is interesting that TLA$^+$ makes such direct use of first order logic, and yet appears not to permit verbatim translation into the specification logic of a theorem prover, whereas BSV, which instead opts for functionally-inspired syntax, permits direct translation into higher order logic by the application of monads.

TLC

TLC [YML99] is model checker and simulator for a subset of TLA$^+$. The specific subset is not clearly defined in the literature, but seems to be essentially the class of finite-state specifications in which transitions are specified in guarded action form.

TLC is custom-built in Java, and uses an explicit model checking algorithm to verify deadlock freedom and safety properties. It does not address module instantiation.

Other Approaches

The Temporal Logic of Actions (TLA) [Lam94] is a formalism which predates TLA$^+$. It is a more primitive language, which does not possess the module structure of its successor – in this sense, it bears only a passing resemblance to BSV. TLA has been embedded in the specification logics of a number of proof tools: the Larch Prover [EGL93]; COSPAN/Larch [KL93]; PVS [SSA01]; Isabelle [Kal95]; HOL [Lå94].

8.2.3 Unity

UNITY is another logic for concurrent systems, which provides a minimal guarded action language together with a fragment of linear temporal logic for expressing safety and liveness properties. Systems are specified with:

1. A single global state;
2. A set of valid initial states;
3. A set of guarded actions which transform the global state.

A simple notion of program composition is provided. Systems which operate on the same global state can be composed by taking the intersection of their sets of initial states and the union of their sets of guarded actions. This is significantly more primitive than the constructs provided by BSV for program composition.

As with TLA and TLA$^+$, UNITY was originally developed as a formalism for hand-proof, but has subsequently been embedded in several proof tools. Paulson [Pau00] embeds UNITY in the higher order logic of the Isabelle theorem prover, using a set-based formalism. Individual guarded actions are specified as binary relations on the state type (as with the formalisation of BSV rules presented in §5.3) and are collected into a set. Together with a state type and a set of initial states, this completes the formalisation of a UNITY program. Composition is defined in terms of set operations, as described above. Programs are verified with interactive deduction, where proof effort is reduced by the use of Isabelle's proof tactics. These include a simplifier which performs conditional, contextual and permutative rewriting and a classical reasoner which applies tableau methods. Heyd and Crégut [HC96] encode

a subset of UNITY in the type theory tool Coq. They achieve composition of safety and progress properties by associating programs with 'contexts', which specify the transitions that can be performed by *other* programs that operate concurrently with the program in question. Andersen *et al.* [APP94] provide an embedding of UNITY in the HOL theorem prover, together with a deductive proof tactic for discharging manageable proof sub-goals. However, they do not consider compositional reasoning. Kaltenbach [Kal96] presents symbolic model checking algorithms for verifying safety and progress theorems concerning finite-state UNITY programs.

8.2.4 Event-B

Event-B is another guarded action formalism for high-level modelling of concurrent systems. In contrast to TLA, TLA+ and UNITY, it does not provide a temporal logic for specifying system properties, but instead focuses on proof of refinement. Systems are described in the standard guarded action format (state type, initial state(s) and guarded actions). In addition to this, language features are provided to express program refinement. One system can be declared as a refinement of another, and individual guarded actions belonging to the former can be declared as refinements of corresponding guarded actions in the latter. The core Event-B language does not provide constructs for modular design, although a 'plug-in' has been created to achieve this [Wik10]. Event-B specifications are developed and refined using the Rodin platform [Rod10, ABHV06]. We do not elaborate on this work, because it is concerned with refinement proof, whereas the present work is concerned with property verification. However, we have seen that BSV supports a refinement based design cycle, which means that the approach taken by Event-B to implement mechanical refinement proof may be of interest for further work.

8.2.5 Languages for Model Checking

As discussed in chapter 4, guarded actions are widely used in model checking formalisms, for a host of stand-alone model checkers such as SAL, SPIN and Murphi, in addition to hybrid model checking and theorem proving tools. As with the hand-based formalisms UNITY, TLA and Event-B, these languages are kept intentionally simple in order to reduce the effort of proof.

The technical challenge in applying automated reasoning to BSV lies to a large extent in the translation of BSV programs to the simpler guarded action

languages which are provided by automated proof tools. In this respect, the technical contributions of the present work are not directly related to the literature of languages for model checking, which are purpose-built to reduce the effort of mechanical verification.

8.3 Functional Hardware Languages

A common theme among semantically elegant hardware languages is the use of functional programming, either to describe circuits directly or to embed hardware languages which, in turn, often have a functional flavour. In fact, BSV is an embedded domain-specific language (EDSL) within Haskell, and also shows strong influences from the functional paradigm in its language features.

BSV is unique among functional and functionally-embedded hardware languages in the fundamental issue of its underlying model of concurrency, meaning that it presents a distinct set of challenges for the application of automated reasoning. For example, *stream transformer languages* (such as Cryptol) specify hardware in terms of functions over infinite streams; this creates a challenge for proof tools, which generally prohibit non-terminating functions. In contrast, BSV designs specify hardware using guarded actions (which allow direct translation to logic, in the form of binary transition relations) but present other challenges, such as a heavy reliance upon side-effects (which we have addressed with an application of monads).

8.3.1 Behavioural Languages

Cryptol is an EDSL within Haskell which allows cryptographic algorithms to be specified and compiled to C, Verilog or VHDL. Systems are described at the behavioural level, as functions which operate on streams of input values. Cryptol was designed by Galois Inc. for the US National Security Agency, to act as a public standard for specifying cryptographic algorithms. Because of its information-critical nature, it is well supported by automated proof tools, which provide two verification approaches [EM09]:

1. For a subset of the language, SAT and SMT techniques implement fully automated safety-checking, as well as proof of equivalence between incrementally refined Cryptol designs, and also between Cryptol and corresponding VHDL.

2. For programs which cannot be verified automatically with SAT and SMT techniques, Cryptol can be compiled to Isabelle/HOL for verification with interactive theorem proving.

In both cases, Cryptol programs are first translated to expressions in a simpler symbolic bit-vector language, which are then translated to the input logic of the proof tool in question.

8.3.2 Structural Languages

A number of functional and functionally-embedded languages define hardware at the structural level: these include Hawk [MCL98], Ruby [She90] and Lava [BCSS98].

Matthews and Launchbury [ML99, Mat00] embed Hawk in the higher order logic of Isabelle. Hawk is a functional language, which extends Haskell with domain-specific features for hardware design. It describes hardware with functions, which Matthews and Launchbury translate directly into logic. A complication arises in formalising recursive functions, which have a domain-theoretic definition in Hawk. This is captured in logic with converging equivalence relations.

A subset of Ruby has also been embedded in Isabelle [Ras96, SR97]. A shallow embedding is formed in Zermelo-Fraenkel set theory (which is well supported in Isabelle) and proofs are performed by structural induction.

Lava [BCSS98] provides automatic compilation of logic specifications and theorems for the propositional tautology checker Prover [Sta89] and the first order theorem provers Otter [MW97] and Gandalf [Tam97]. The theorems can then be proven to establish equivalence between designs.

8.3.3 Synthesis from Logic

A number of investigations have been made into the feasibility of synthesising hardware from functional descriptions written in the specification logics of theorem provers. These include: LAMBDA (Logic And Mathematics Behind Design Automation) [FFFH89]; VERITAS$^+$ [HDL89]; Gropius [Blu99]; HOL synthesis [GIOS06].

8.4 Automated Reasoning for *Ad Hoc* Languages

Automated reasoning has been used extensively to verify designs in the common *ad hoc* hardware languages, but its effectiveness is impaired by two factors:

1. **Convoluted Semantics** – languages such as VHDL, Verilog and SystemC have convoluted *simulation cycle semantics* [BGG+92, Gor95, Klo95, MRH+01], which complicate the task of translating programs into tractable logic expressions.

2. **Unrelated System Models** – design flows commonly produce two system models: a concise ESL specification and a synthesizable RTL implementation. These are often written in different languages, and constructed independently, rather than being linked by a process of refinement.

In contrast, BSV has an intentionally elegant semantics, and supports a design flow in which ESL specifications are incrementally refined until designs are produced which can be synthesised to efficient hardware.

There is an extensive body of literature on the application of automated reasoning to *ad hoc* hardware languages. However, because it is tangential to the present work, we restrict our discussion to four of the larger projects – undertaken by Intel, AMD, IBM and Centaur – in order to outline the central themes of automated reasoning for this style of language. Further to this, it is informative to gain a picture of how automated reasoning is applied by the big players in IC design.

The aforementioned companies have independently developed proof tools for RTL verification, which have a number of similarities:

1. They verify RTL code against high-level logic specifications.

2. RTL code is first translated to a semantically elegant state machine language, which is then embedded in logic.

3. High-level logic specifications are formed by manually translating informal requirement specifications and ESL specifications.

4. Intel and Centaur use compositional reasoning, where automatic techniques are used to efficiently verify low-level temporal logic properties, which are then used during interactive deduction to prove correctness against a high-level logic specification.

8.4.1 Forte

The Forte verification system [SJO+05] combines model checking with lightweight theorem proving in higher order logic, in order to verify RTL designs. It is owned

and developed by Intel, although it is freely available for non-commercial use. Forte evolved from earlier work [JS93] on combining the HOL theorem prover [GM93] with the VOSS symbolic trajectory evaluation (STE) model checker [Seg93]. RTL is translated to gate-level descriptions, which are automatically translated to a finite state machine (FSM) representation in FL [AJS99], a strongly typed, higher-order functional programming language. The STE model checker is then used to verify these finite state machines against low-level temporal logic properties, which are used during interactive proof to verify that the FSMs are complicit with high-level logic specifications. The STE model checker can verify large design fragments for a limited temporal logic, which includes the 'next state' and 'requires' path quantifiers (§4.4). State space reduction is achieved by using a three valued logic, where digital signals can be 'high', 'low' or 'don't care'; this can dramatically reduce the number of cases that must be considered by the model checker. Successful industrial-scale applications include [AJS98, KA00, OZGS99].

8.4.2 RuleBase

IBM has produced the RuleBase symbolic model checker [BDEGW03], which is based on the Symbolic Model Verifier (SMV) [McM93], and makes extensive use of state space reduction techniques, including cone of influence reduction, flip-flop equivalence, constant propagation and abstraction-refinement. RuleBase supports two design flows:

1. System level specifications can be produced in a subset of Java, and verified against temporal logic specifications [ESH+00, AVARB+01]. RTL code can then be automatically synthesised from the Java specification.

2. RTL can be hand-coded as part of a conventional design flow, and then verified directly against temporal logic specifications.

Properties are expressed in a temporal logic called the Property Specification Language (PSL) [EF06], which provides CTL and LTL operators, along with user-friendly syntactic sugar. Six separate back-ends are available for verification, each based on a different algorithmic approach, using binary decision diagrams (BDDs), SAT-solving and semi-formal techniques.

RuleBase has been widely used at IBM, and also at a number of other companies including STMicroelectronics, Galileo Technology, Mellanox Technologies and Zoran

Corporation. Successful industrial-scale applications include [BH98, AVARB+01, BHSA99, GL00, Par00].

8.4.3 DE2

Centaur Technology and the University of Texas have developed an approach for verifying Verilog circuits with the ACL2 first order theorem prover. Verilog designs are automatically translated to equivalent descriptions in a semantically elegant state machine language called DE2, which has been deeply embedded in the ACL2 logic [HR05]. During the Verilog-to-DE2 translation, cone of influence reduction is applied to reduce the state space. High-level ACL2 specifications are manually constructed from ESL designs and informal requirement specifications.

The DE2 system achieves scalable verification in much the same way as Forte. Finite state machines specified in DE2 are automatically verified against low-level temporal logic assertions, which are used during interactive proof to establish complicity with high-level ACL2 specifications. Centaur has used its system to verify floating-point addition and subtraction instructions for the media unit of a 64-bit, X86-compatible microprocessor [HS09].

8.4.4 AMD

AMD has also employed ACL2, for the verification of hardware designs expressed in its proprietary RTL language [Obe99, RF00, Rus00b], in an approach which bears a strong resemblance to Forte and the DE2 system. RTL is translated to a Lisp-like finite state machine language with clean semantics, which is embedded in the ACL2 logic. The resulting ACL2 specifications are verified with interactive proof, using mathematical induction, conditional rewriting and decision procedures. This approach has been used to verify floating point algorithms which are used in the AMD Athlon processor and its derivatives.

8.5 Summary

A survey has been conducted of the literature which applies automated reasoning to: monadic specifications, guarded action languages, functional and functionally-embedded hardware languages, and *ad hoc* hardware languages. We have seen that BSV is a unique language when considered with respect to automated reasoning, being

unusually complex for a guarded action language, and exceptional amongst hardware languages in its choice of guarded actions as the underlying model of concurrency.

Chapter 9

Conclusion

The themes and contributions of this dissertation are summarised, and a discussion is presented of possible topics for further work.

9.1 Concurrent Haskell

A novel approach has been presented for applying lightweight formal methods to packet-switched networks-on-chip, which involves behavioural specification in Concurrent Haskell, and verification by simulation and hand-proof. This approach has been used to specify and verify a novel NoC architecture belonging to the SpiNNaker many-core processor.

This work was motivated by a requirement of the SpiNNaker design team for rigorous verification of the system-wide communications network, which contains a myriad of cyclic paths, as well as a novel emergency routing protocol that spontaneously redirects packets in real-time. The combination of these two features created a tangible concern over the possibility of livelock. A behavioural specification has been developed for the SpiNNaker NoC, which focusses in particular on the novel emergency routing mechanism. Furthermore, a hand proof has been presented which verifies an aspect of the emergency routing protocol with mathematical induction.

It has been shown that Concurrent Haskell provides a suitable basis for the application of lightweight formal methods to NoC verification. Owing to its clean semantics, it allows the construction of elegant behavioural specifications which can be executed (Concurrent Haskell is a mature software language) and also subjected to formal reasoning with hand proof.

9.2 Bluespec SystemVerilog

A subset of Bluespec SystemVerilog has been embedded in the PVS theorem prover and the SAL model checker. The subset which has been embedded includes module definition and instantiation, methods, implicit conditions, scheduling attributes, and rule composition using methods from instantiated modules.

In the case of PVS, a novel application of monadic techniques led to a surprisingly elegant embedding, in which BSV is translated into logic almost verbatim, preserving types and language constructs. Proof strategies have been presented in the PVS strategy language, which automatically verify an important class of temporal logic theorems concerning instances of the monadic embedding strategy, using a combination of model checking and automated deduction.

The SAL language was found to permit monadic programming. However, whilst monadic BSV specifications are accepted by the SAL type checker, the various model checking back-ends fail to reduce them to a tractable form. Despite this, SAL was found to be compatible with a simpler 'primitive' embedding strategy, in which BSV constructs such as module instantiation and method invocation are eliminated by in-place expansion. This raises the possibility of hybrid verification, in which primitive specifications are model checked by SAL, and other proof techniques, such as automatic abstraction and compositional reasoning, are applied with a tool such as PVS.

The contributions of this dissertation represent a step in the journey towards scalable automated reasoning for BSV. Model checking, together with a small amount of automated deduction, was sufficient to verify the BSV examples considered, whereas more sophisticated proof techniques will be required to verify designs of real-world complexity. However, before automated reasoning can be applied to any BSV design, it must first be translated into the logic of an appropriate proof tool, and the choices of both tool and translation strategy can have a significant impact on the applicability of proof techniques. It has been shown here that a considered approach to embedding, in a versatile proof tool such as PVS, produces surprisingly elegant specifications which can be directly verified with a combination of model checking and deductive reasoning, whilst also being apparently well-disposed for extension in further work with automatic predicate abstraction, compositional reasoning and a host of other proof techniques. In this respect, the present work has aimed to make a first step which, however modest, is a step in the right direction.

9.2.1 Topics for Further Work

The embeddings of BSV in PVS and SAL create a number of possibilities for further research. PVS, in particular, was chosen because of its seamless integration of model checking, automatic abstraction and deductive reasoning. This dissertation, however, has only scratched the surface of its potential for BSV verification. A natural next step would be the application of automatic predicate abstraction to monadic BSV specifications. This facility was developed in PVS to complement its model checker [SS99], and the two proof techniques can, in fact, be applied with a single invocation of the proof strategy `(abstract-and-mc)` [SORSC01]. An application of automatic abstraction could dramatically increase the class of BSV designs that can be efficiently verified within PVS.

Compositional reasoning is another avenue for investigation. Monadic embedding maintains the clean partitioning of state and functionality from the original source code, which raises the possibility of applying compositional reasoning to BSV designs, essentially at the source code level. Temporal logic properties could be automatically proven for the methods of small modules (using model checking with abstraction) and composed by user-guided deduction to verify the behaviour of more complex modules which invoke the methods. Intel's Forte tool (§4.7.2, §8.4.1), for example, makes extensive use of compositional reasoning, although the technical difficulty of formalising RTL necessitates a more contrived path, in which RTL is first compiled to the gate-level, before low-level temporal properties are verified with STE model checking and composed by interactive deduction to verify higher-level properties.

Automatic BSV-to-logic translation would provide a useful building block for further work, and might also serve to stimulate new research in the area, because it would lower the technical barrier for experimentation with automated reasoning for BSV. Experimentation in this dissertation with the PVS strategy language suggests that compilation of BSV to monadic PVS could be accompanied by the automatic generation of proof strategies, which would further increase the usability of such a tool. It would be interesting to see how the concept of strategy creation will scale as more sophisticated verification approaches are considered.

The concept of extensional equivalence between monadic and primitive embeddings has another potential application: construction of a verifying compiler. BSV compilation begins with a reduction of the source code to a simpler 'abstract transition system' (ATS) form, which is similar to the expressions generated by the primitive

embedding strategy proposed in the present work. It might be possible to simultaneously compile BSV source code to monadic form in PVS, whilst also compiling the corresponding abstract transition system to a primitive form. Extensional equivalence could then be proven between the two PVS specifications, thus verifying the BSV-to-ATS compilation. Proof of extensional equivalence between monadic and primitive specifications can be achieved automatically in PVS, with a single application of the strategy `(grind-with-ext)`. It may be possible to employ a SAT or SMT solver (§4.5) for this task, in order to produce a fully automatic, command line driven verifier for BSV-to-ATS compilation.

The present work has produced encouraging results with PVS and SAL, although both tools have their limitations. SAL is an effective model checker, but does not support proof techniques such as abstraction and deduction, which seem to be essential for the verification of large-scale BSV designs. PVS, on the other hand, provides close integration of a broad range of proof techniques, but its model checker is somewhat unrefined (§6.1.1). This may warrant a thorough evaluation of alternative tools, which were discussed in §4.7.2. An obvious alternative to PVS is HOL, which also supports higher order theorem proving with integrated model checking and automatic abstraction. HOL is a mature tool which is being actively developed at the University of Cambridge, and is employed in a number of academic institutions.

This dissertation has investigated one area of an expansive – and mostly unexplored – research space concerning automated reasoning for Bluespec SystemVerilog. BSV programs are naturally amenable to expression in formal logic, which raises the possibility of relatively painless experimentation with a broad range of automated formal and semi-formal methods. Possibilities for further work include:

- **Semi-formal methods:** for example, automatic test pattern generation or reachability analysis. An interesting discussion of semi-formal methods can be found in [BAWR07].

- **Automatic refinement proof** from ESL specifications down to efficiently synthesizable implementations – perhaps based on Event-B style refinement (§8.2.4).

- **Evaluation of deep and reflective embeddings**, to investigate the utility of reasoning at the language level (as discussed in §5.12) and also to compare the efficiency of automated reasoning for BSV programs expressed using shallow, deep and reflective embedding strategies.

9.3 Final Thoughts

Formal methods play a significant role in hardware verification, and their presence is set to increase as designs exhibit greater complexity and traditional verification techniques become progressively less effective. In 2009, ITRS reported that 9.4% of design errors in the companies it surveyed were identified with formal or semi-formal methods, and stipulated that this should increase to 45% over the following 15 years [ITR09].

One impediment to the achievement of this target is the *ad hoc* nature of mainstream design languages – such as VHDL, Verilog and SystemC – which have evolved organically as the industry has matured, but now seem ill-suited to the oncoming influx of formal methods. This thesis investigated an alternative class of *semantically elegant* languages, which allow efficient design and synthesis, whilst having also been developed with formal reasoning in mind.

A particularly promising result has been the application of automated reasoning to the semantically elegant language Bluespec SystemVerilog. BSV has already been shown to increase design efficiency by reducing design time when compared to hand-written VHDL or Verilog, whilst producing comparable hardware for many applications [GW08, Nik04, WNRD04]. In addition to this, the present work has shown that it also provides an efficient basis for the application of automated reasoning.

To some extent, the notion of migrating towards semantically elegant languages is revolutionary, in an industry which favours evolution. Traditional design and verification techniques have become entrenched, and the cost of transition to alternative methodologies will be high. However, IC production is a highly competitive activity, where the commercial advantage is gained by realising sophisticated designs at the lowest cost. With Moore's law consistently out-pacing improvements in design efficiency, design non-recurring engineering costs now frequently exceed manufacturing non-recurring engineering costs by an order of magnitude, meaning that production costs are increasingly determined by the metric of design efficiency [ITR09]. As we have seen, ITRS predicts that increasing design efficiency will be achieved over the coming decades from innovation in the application of formal methods.

Bibliography

[ABHV06] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. 2006.

[AH07] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[AJS98] M. Aagaard, R. Jones, and C.-J. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Proceedings of the 35th annual Design Automation Conference*, pages 538–541. ACM, 1998.

[AJS99] M. Aagaard, R. Jones, and C.-J. Seger. Lifted-FL: A pragmatic implementation of combined model checking and theorem proving. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 323–340. Springer-Verlag, 1999.

[Amj03] H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics, volume 2758 of Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2003.

[Amj06] H. Amjad. Verification of AMBA using a combination of model checking and theorem proving. *Electronic Notes in Theoretical Computer Science*, 145:45–61, 2006.

[APP94] F. Andersen, K. Petersen, and J. Pettersson. Program verification using HOL-UNITY. In *Proceedings of the 6th International Workshop on*

Higher Order Logic Theorem Proving and its Applications, pages 1–15. Springer-Verlag, 1994.

[AS99] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3), 1999.

[AVARB+01] Y. Abarbanel-Vinov, N. Aizenbud-Reshef, I. Beer, C. Eisner, D. Geist, T. Heyman, I. Reuveni, E. Rippel, I. Shitsevalov, Y. Wolfsthal, and T. Yatzkar-Haham. On the effective deployment of functional formal verification. *Formal Methods in System Design*, 19(1):35–44, 2001.

[BAWR07] J. Bhadra, M.. Abadir, L.-C. Wang, and S. Ray. A survey of hybrid techniques for functional verification. *IEEE Design and Test of Computers*, 24(2):112–122, 2007.

[BBC+95] N. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z Manna, H. Sipma, and T. Uribe. STeP: The stanford temporal prover (educational release) user's manual. Technical report, 1995.

[BBC+96] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 415–418. Springer-Verlag, 1996.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320, New York, NY, USA, 1999. ACM.

[BCSS98] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[BDD07] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: an extensible automated theorem prover producing checkable proofs. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 151–165. Springer-Verlag, 2007.

[BDEGW03] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at IBM. *Formal Methods in System Design*, 22(2):101–108, 2003.

[BF02] J. Bainbridge and S. Furber. Chain: A delay-insensitive chip area interconnect. *IEEE Micro*, 22(5):16–23, 2002.

[BFN+06] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design*, pages 129–156. North-Holland Publishing Co., 1992.

[BH98] J. Baumgartner and T. Heyman. An overview and application of model reduction techniques in formal verification. In *Proceedings of the IEEE International Conference on Performance, Computing and Communications*, pages 165–171, 1998.

[BHPS07] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A generic model for formally verifying NoC communication architectures: A case study. *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pages 127–136, May 2007.

[BHSA99] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. Model checking the IBM gigahertz processor: An abstraction algorithm for high-performance netlists. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 72–83. Springer-Verlag, 1999.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2^{nd} edition, 1998.

[BKH+08] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In *Proceedings*

of the 21st International Conference on Theorem Proving in Higher Order Logics. Springer-Verlag, 2008.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Invest: A tool for the verification of invariants. In *Computer Aided Verification, volume 1427 of LNCS*, pages 505–510. Springer-Verlag, 1998.

[Blu99] C. Blumenröhr. A formal approach to specify and synthesize at the system level. In *GI Workshop Modellierung und Verifikation von Systemen*. Shaker-Verlag, 1999.

[Blu08] Bluespec, Inc. *Bluespec SystemVerilog Reference Guide*, 2008.

[BM08] P. Böhm and T. Melham. A refinement approach to design and verification of on-chip communication protocols. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 2008.

[BSVMH84] R. Brayton, A. Sangiovanni-Vincentelli, C. McMullen, and G. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[CB00] R. Colwell and R. Brennan. Intel's formal verification experience on the Willamette development. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 106–107. Springer-Verlag, 2000.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499. Springer-Verlag, 1999.

[CDLM08] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA+ proof system. In *Proceedings of the LPAR Workshop Knowledge Exchange: Automated Provers and Proof Assistants*, 2008.

[CDLM10] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148. 2010.

[CGP00] E. Clarke, Jnr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[Chu40] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[CM88] K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[DF95] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 54–69. Springer-Verlag, 1995.

[dM04] L. de Moura. SAL: Tutorial. Technical report, SRI International, November 2004.

[DM06] Bruno D. and L. De Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[DMB08] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer-Verlag, 2008.

[dMOR+04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proceedings of the 16th International Conference on Computer Aided Verification*, 2004.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer-Verlag New York, Inc., 2006.

[EGL93] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In *Proceedings of the First International Workshop on Larch*, pages 86–97. Springer-Verlag, 1993.

[EM09] L. Erkök and J. Matthews. Pragmatic equivalence and safety checking in Cryptol. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*. ACM, 2009.

[ESH+00] C. Eisner, I. Shitsevalov, R. Hoover, W. Nation, K. Nelson, and K. Valk. A methodology for formal design of hardware control with application to cache coherence protocols. In *Proceedings of the 37th Annual Design Automation Conference*, pages 724–729. ACM, 2000.

[FFFH89] S. Finn, M. Fourman, M. Francis, and R. Harris. Formal system design – interactive synthesis based on computer-assisted formal reasoning. In *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[Fil03] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4), 2003.

[FM10] A. Fox and M. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer Berlin / Heidelberg, 2010.

[FSNB09] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(1):27–45, 2009.

[FSS+94] T. Filkorn, H. Schneider, A. Scholz, A. Strasser, and P. Warkentin. SVE user's guide. Technical Report ZFE BT SE 1-SVE-1, 1994.

[FT08] S. Furber and S. Temple. Neural systems engineering. In *Computational Intelligence: A Compendium*, volume 115 of *Studies in Computational Intelligence*, pages 763–796. Springer, 2008.

[FTB06a] S. Furber, S. Temple, and A. Brown. High performance computing for systems of spiking neurons. In *Proceedings of the Workshop on Adaptation in Artificial and Biological Systems*, 2006.

[FTB06b] S. Furber, S. Temple, and A. Brown. On-chip and inter-chip networks for modeling large-scale neural systems. In *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems*. IEEE Computer Society Press, 2006.

[GDR05] K. Goossens, J. Dielissen, and A. Radulescu. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test of Computers*, 22:414–421, September 2005.

[GIOS06] M. Gordon, J. Iyoda, S. Owens, and K. Slind. Automatic formal synthesis of hardware from higher order logic. *Electronic Notes in Theoretical Computer Science*, 145:27–43, 2006.

[GL00] A. Goel and W. Lee. Formal verification of an IBM CoreConnect processor local bus arbiter core. In *Proceedings of the 37th Annual Design Automation Conference*, pages 196–200. ACM, 2000.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GMO06] J. Grundy, T. Melham, and J. O'leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, page 142. IEEE Computer Society, 2002.

[Gor95] M. Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, page 136. IEEE Computer Society, 1995.

[GPB01] E. Goldberg, M. Prasad, and R. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 114–121. IEEE Press, 2001.

[Gro02] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83. Springer-Verlag, 1997.

[GVZ+05] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, Edwin Rijpkema, and Andrei R. Deadlock prevention in the Æthereal protocol. In *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 345–348. Springer Berlin / Heidelberg, 2005.

[GW08] F. Gruian and M. Westmijze. VHDL vs. Bluespec SystemVerilog: a case study on a Java embedded architecture. In *Proceedings of the 2008 ACM Symposium on Applied Computing*. ACM, 2008.

[Har09] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[Has11] Haskell.org. Haskell standard library: Control.concurrent.chan. http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Concurrent-Chan.html, 2011.

[HC96] B. Heyd and P. Crégut. A modular coding of UNITY in COQ. In *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin / Heidelberg, 1996.

[HDL89] F. Hanna, N. Daeche, and M. Longley. Formal synthesis of digital systems. In *IFIP International Workshop on Applied Formal Methods for Correct VLSI Design.*, 1989.

[HJvdB01] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's vector class. *International Journal on Software Tools for Technology Transfer*, 3:332–352, 2001.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *A T & T Technical Journal*, 69(1):44 – 59, 1990.

[HMW05] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin / Heidelberg, 2005.

[Hol03] G. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, 2003.

[HR05] Warren A. Hunt and Erik Reeber. Formalization of the DE2 language. In *Proceedings of the 13th Working Conference on Correct Hardware Design and Verification Methods*, pages 20–34. Springer-Verlag, 2005.

[HS09] W. Hunt and S. Swords. Centaur technology media unit verification. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 353–367. Springer-Verlag, 2009.

[IEE94] IEEE. IEEE standard VHDL language reference manual: ANSI/IEEE std 1076-1993. Technical report, 1994.

[Int10] Intel Corp. Intel® Core™ i7-900 desktop processor extreme edition series and Intel® Core™ i7-900 desktop processor series specification update, January 2010.

[ITR09] ITRS. *International Technology Roadmap for Semiconductors, 2009 Edition*, chapter Design. 2009.

[JP00] B. Jacobs and E. Poll. A monad for basic Java semantics. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*. Springer-Verlag, 2000.

[JS93] J. Joyce and C.-J. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Proceedings of the 30th international Design Automation Conference*, pages 469–474. ACM, 1993.

[KA00] R. Kaivola and M. Aagaard. Divider circuit verification with model checking and theorem proving. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 338–355. Springer-Verlag, 2000.

[Kal95] S. Kalvala. A formulation of TLA in Isabelle. In *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 214–228. 1995.

[Kal96] M. Kaltenbach. *Interactive verification exploiting program design knowledge: a model-checker for UNITY*. PhD thesis, 1996.

[KJS+02] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings. IEEE Computer Society Annual Symposium on VLSI*. IEEE Computer Society Press, 2002.

[KL93] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *Proceedings of the 5th International Conference on Computer Aided Verification*, pages 166–179. Springer-Verlag, 1993.

[Klo95] C. Kloos. *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[KM02] S. Krstic and J. Matthews. Verifying BDD algorithms through monadic interpretation. In *Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 317–320. Springer Berlin / Heidelberg, 2002.

[KMN+00] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523–1543, 2000.

[KSM96] M. Kaufmann and J. Strother Moore. ACL2: an industrial strength version of Nqthm. *Proceedings of the Eleventh Annual Conference on Computer Assurance*, pages 23–34, Jun 1996.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[L̊94] T. Långbacka. A HOL formalisation of the temporal logic of actions. In *Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 332–345. 1994.

[Mat00] J. Matthews. *Algebraic Specification and Verification of Processor Microarchitectures*. PhD thesis, 2000.

[MCL98] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE Computer Society, 1998.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mil93] R. Milner. *Logic and Algebra of Specification*, chapter The polyadic pi-calculus: a tutorial, pages 203–246. Springer-Verlag, 1993.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[ML99] J. Matthews and J. Launchbury. Elementary microarchitecture algebra. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 288–300. Springer-Verlag, 1999.

[MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 1–16. Springer-Verlag, 1995.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE Press, 1989.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Moo65] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[MRH+01] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of SystemC. In *Proceedings of the conference on Design, automation and test in Europe*, pages 64–70. IEEE Press, 2001.

[MW97] William McCune and Larry Wos. Otter. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[Nik04] R. Nikhil. Bluespec SystemVerilog: efficient, correct RTL from high level specifications. In *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*. IEEE Press, 2004.

[NWP02] T. Nipkow, M. Wenzel, and L. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, 2002.

[Obe99] S. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, page 106. IEEE Computer Society, 1999.

[OGS08] B. O'Sullivan, J. Goerzen, and . Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2008.

[ORR+96] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Berlin / Heidelberg, 1996.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752. Springer-Verlag, 1992.

[OSRSC01] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. PVS language reference. Technical report, SRI International, 2001.

[Owr08] S. Owre. PVS 3.2 release notes: grind-with-ext and reduce-with-ext. http://pvs.csl.sri.com/pvs-release-notes/pvs-release-notes_4.html#SEC37, 2008.

[OZGS99] J. O'Leary, X. Zhao, R. Gerth, and C.-J. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, (Q1):10, February 1999.

[Par00] A. Parash. Formal verification of an MPEG decoder chip - a case study in the industrial use of formal methods. In *Proceedings of the Workshop on Advances in Verification*, 2000.

[Pau00] L. Paulson. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic*, 1:3–32, July 2000.

[Pet81] J. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[Pie02] B. Pierce. *Types and programming languages*. MIT Press, 2002.

[PJGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM, 1996.

[PTC+00] V. Pisini, S. Tahar, P. Curzon, O. Ait-Mohamed, and X. Song. Formal hardware verification by integrating HOL and MDG. In *Proceedings of the 10th Great Lakes Symposium on VLSI*, pages 23–28. ACM, 2000.

[Ras96] O. Rasmussen. An embedding of Ruby in Isabelle. In *Proceedings of the 13th International Conference on Automated Deduction*, pages 186–200. Springer-Verlag, 1996.

[RF00] D. Russinoff and A. Flatau. RTL verification: a floating-point multiplier. pages 201–231, 2000.

[RL11] D. Richards and D. Lester. Source code for primitive and monadic embeddings of BSV in PVS and SAL. https://sourceforge.net/projects/ar4bluespec, 2011.

[Rod10] Rodin. Rodin project homepage. http://rodin.cs.ncl.ac.uk/index.htm, 2010.

[RSS95] S. Rajan, N. Shankar, and M. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the 7th International*

Conference on Computer Aided Verification, pages 84–97. Springer-Verlag, 1995.

[Rus00a] J. Rushby. From refutation to verification. In *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX)*. Kluwer, B.V., 2000.

[Rus00b] D. Russinoff. A case study in fomal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 3–36. Springer-Verlag, 2000.

[SB07] C. Sprenger and D. Basin. A monad-based modeling and verification toolbox with application to security protocols. In *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 302–318. Springer Berlin / Heidelberg, 2007.

[SB08] J. Schmaltz and D. Borrione. A functional formalization of on chip communications. *Formal Aspects of Computing*, 20:241–258, May 2008.

[Seg93] C. Seger. VOSS - a formal hardware verification system user's guide. Technical report, 1993.

[She90] M. Sheeran. Describing butterfly networks in Ruby. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag, 1990.

[Sim07] R. Siminiceanu. Model checking in SAL. In *LaRC PVS Class*. National Institute of Aerospace, November 2007.

[SJO+05] C.-J. Seger, R. Jones, J. O'Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[SN94] K. Stroetmann and C. Nielsen, editors. *A Guide to SEDUCT*. Siemens AG, 1994.

[SORSC01] N. Shankar, S. Owre, J. Rushby, and D. Stringer-Calvert. PVS prover guide. Technical report, SRI International, 2001.

[SR97] R. Sharp and O. Rasmussen. The T-Ruby design system. *Formal Methods in System Design*, 11:239–264, 1997.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 443–454. Springer-Verlag, 1999.

[SS08] G. Singh and S. Shukla. Verifying compiler based refinement of Bluespec specifications using the SPIN model checker. In *Proceedings of the 15th International Workshop on Model Checking Software*, pages 250–269. Springer-Verlag, 2008.

[SSA01] J. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*. Springer-Verlag, 2001.

[Sta89] G. Stalmarck. *A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula*. Swedish Patent No. 467 076, 1989.

[Tam97] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[TM96] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1996.

[Wad92a] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:61–78, 1992.

[Wad92b] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.

[Wik10] Event-B Wiki. Event-B D23 modularisation plug-in. http://wiki.event-b.org/index.php/D23_Modularisation_Plug-in, 2010.

[WNRD04] W. Wong, R. Nikhil, D. Rosenband, and N. Dave. High-level synthesis: an essential ingredient for designing complex ASICs. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, pages 775–782. IEEE Computer Society, 2004.

[YML99] Y Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer-Verlag, 1999.